# Department of Computer Science


# Technical Report

Undergraduate Dissertation: Facilitating the Creation of Advanced Agents within NetLogo by Allowing Specification and Control Using the Behaviour Oriented Design Methodology

Michael Brooks

**Technical Report 2013-07** **November 2013**

**ISSN 1740-9497**

**Contact Address:**
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

**ISSN 1740-9497**

# Facilitating the Creation of Advanced Agents within NetLogo by Allowing Specification and Control Using the Behaviour Oriented Design Methodology

Michael Brooks

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2013

# Facilitating the Creation of Advanced Agents within NetLogo by Allowing Specification and Control Using the Behaviour Oriented Design Methodology

Submitted by: Michael Brooks

## COPYRIGHT



## Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

# Acknowledgements

Firstly I would like to thank Joanna Bryson, my supervisor, for her patience and help throughout the project. Also, my thanks to the many people who have worked on the various POSH schedulers, and BOD in general. Though there are too many to mention here, I would like to mention Swen Gaudl in particular, who provided me with a scheduler that provided the foundations for my own version, as well as willingly answering my questions.

Finally, my friends and family for their support and understanding which made this whole process manageable.

**Abstract**

NetLogo (Wilensky, 1999) is a very popular agent based modelling platform that is commonly used in a wide range of scientific fields. Behaviour Oriented Design (Bryson, 2003*a*) is a development methodology for creating complex agents, it uses a form of action selection known as POSH (Parallel-Rooted, Ordered Slip-Stack Hierarchical) as an arbitrator to control the 'external' actions of an agent. This project aims to allow the creation of BOD agents within NetLogo by implementing POSH for NetLogo and providing an example of the design methodology. The final product of the project is BODNetLogo a program which successfully allows the specification of BOD agents which can then be run inside a NetLogo simulation.

# Contents

# List of Figures

# Chapter 1

# Introduction

This project aims to facilitate the creation of complex agents within the agent based modelling platform, NetLogo. To do this I will integrate it with the behaviour oriented design methodology which although based in robotics has a growing presence in virtual reality and game AI platforms, due to its strengths in creating complex agents. Although NetLogo is intended to be a largely educational platform it has grown widely in popularity and is used in many different scientific fields. The simplified programming language however limits the potential of the agents, by integrating with BOD I believe that I can maintain the simplicity and ease of use whilst greatly increasing the potential.

In this chapter I will introduce some key terms as well as the behaviour oriented design methodology and its key ideas and benefits, I will also introduce NetLogo and Agent Based Modelling in general, before finally introducing my proposal to combine the two concepts.

## 1.1 Key Terms

Throughout this report I will be using several key terms whose definition whilst not necessarily complex may be often misunderstood or used in different ways. To ensure these issues do not affect understanding of this report here I provide a definition of the words or terms as they will be used in this report:

- Agent: an agent is anything which can have an effect on its environment. This can include anything from an animal or person to a robot or subject in a simulation. It is worth noting that the NetLogo term *turtle* can be considered to be virtually the same as an agent, as can *patch* although the latter is a rather specific and limited version.
- Complex Agent: a complex agent is an agent which has multiple conflicting goals which it must manage in order to function. For example a robot that needs to explore but also needs to preserve energy and recharge when necessary could be considered to have a goal to move around as much as possible to explore, but also may sometimes

need to be stationary at charging points for hours. The conflicts in goals make the agent complex.

- Complete Complex Agent (CCA): a complete complex agent is a form of complex agent. The term complete indicates that the agent is able to function independently of any multi-agent simulation (MAS). Note this does not mean that any reference to a complex agent implies membership of a MAS, instead it is used to differentiate when discussing fields frequently associated with MAS.
- Reactive Intelligence: reactive intelligence is a term which is often misunderstood to be something which only responds to stimuli, however I will use it to mean intelligence which uses the world 'as its own model' (Brooks, 1991*a*). Put simply the agent does not attempt to work out a perfect plan and then perform it. Instead it continuously checks the world for new information and basis its actions on the current state it finds. This is often known as dynamic intelligence.
- POSH Primitives: The primitives of a posh plan are the 'actions' and 'senses' that are the leaves of the plan and allow the action selection mechanism to interface with the behaviour library.

## 1.2 BOD

Behaviour Oriented Design (BOD) is made of two main components, the BOD architecture and the BOD development process (Bryson, 2001). The development process can be summarised as an iterative design methodology, similar to those in the field of object oriented design (OOD) (Bryson and McGonigle, 1998), and is intended to facilitate the creation of complex agents. The BOD architecture can be further separated into two major parts, a behaviour library and an action plan.

### 1.2.1 Behaviour Library

The behaviours of the agent can be said to control *what* the agent will do, including its actions and senses but also any learning, memory or any other complex calculations. This library is usually written in a traditional OOD programming language (Bryson, 2003*a*). Actions and senses are written as methods within the behaviour or *Object*.

### 1.2.2 Action Selection

The second part of BOD, the action plan, controls 'when' an agent should do something. Specified in the form of a POSH plan (Parallel-Rooted, Ordered, Slip-Stack Hierarchical) (Bryson and Stein, 2001) and interpreted by an *action scheduler* (Bryson, 2001). The action selection mechanism uses the actions and senses, which are known as the primitives of the

plan, as an interface to the behaviours. These primitives are required to always produce an instant (or near instant) response, making the agent reactive and responsive.

POSH plans can be written be hand in a form which is derived largely from the LISP programming language. Or they can be generated using the ABODE IDE which uses a visual representation of the tree structure to make creation and modification easier (See fig. 1.1).

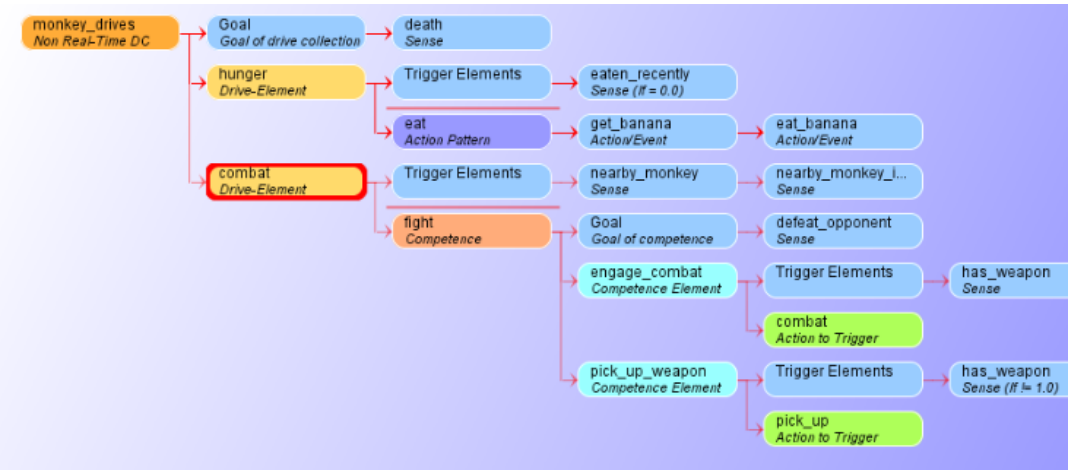


Figure 1.1: A POSH plan in the ABODE IDE. The visual representation of the plan shows the hierarchical nature of the various plan elements. The top level container, the drive collection, is on the far left with drive elements being the next column.

### 1.2.3 Reuse

The separation or *abstraction* of the often complex behaviour code and the basic plans allows re-use of the behaviour libraries in many different plans and potentially by many different planners.

Additionally the hierarchical nature of the plans allows for reuse of plan elements across agents with similar abilities. As each plan element is independent of others it is possible to add in new elements or radically modify existing ones without affecting the rest of the plan.

## 1.3 NetLogo

Agent Based Modelling (ABM) is an idea that is used in a wide range of scientific and business fields (Railsback, Lytinen and Jackson, 2006). It provides a method to run a large

number of simulations that can provide empirical data, much more easily than real world experiments. However, a model is not equivalent to the real world, instead models often depend on assumptions, estimates and simplified concepts. So the gathered data can only be used to support the theory of the experiment and thus act as a prediction about the real world effect being modelled. It is not sufficient evidence to prove a theory in reality. Despite this ABMs are a valuable tool used by many to develop and support theories, particularly when real tests are difficult or expensive to perform.

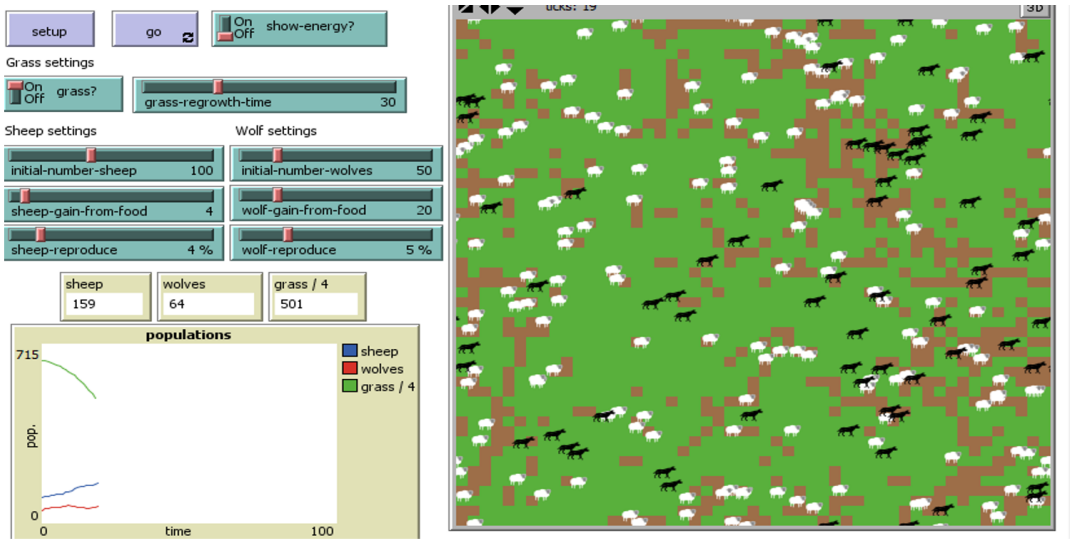


Figure 1.2: The main NetLogo user interface, on the right is the visual representation of the agents inside the world, in this case sheep and wolves with green patches indicating grass and brown patches grass that has been eaten recently by a sheep. On the left there are various buttons and sliders which control the model along with monitors and graphs which show the resulting statistics. All the elements on this screen are easily modified and created using a drag and drop system.

NetLogo (Wilensky, 1999) is an open source ABM platform that allows the simulation and modelling of complex environments and interactions between different 'turtles' or agents (*NetLogo User Manual*, v5.0.3).

Agents and their environment are specified via the customised programming language. A simulation is then run by creating the specified environment and agents, before asking each agent to take a 'turn' and perform their specified actions. One of the reasons NetLogo has become so popular aside from its simple programming language is that it provides a very easy to use and create graphical interface, where users can drag and drop buttons, sliders, graphs and other tools to help control and monitor their simulations (See fig. 1.2).

## 1.4 Project Objectives

The primary objective of this project is to integrate BOD with NetLogo. This has the combined benefits of firstly, allowing more complex agents to be specified within NetLogo, and secondly providing a broader audience for BOD, by increasing its usefulness. Specifically this integration requires the creation of a new, or modification of an existing POSH scheduler to control the BOD agents. Also, in order to demonstrate both the design methodology and the behaviour library structure of BOD, I will create an example project that whilst not having any real scientific value will provide a model for future projects.

Thus, I can briefly describe the functionality required for this integration (See chapter 3 for more details) as a program which allows the control of an agent using a behaviour library written within NetLogo and with an action plan specified in the POSH format. Additionally the program should preserve as much of NetLogo's interface as possible to maintain the ease of use that has led to its popularity.

### 1.4.1 Road Map

The content of this report, and thus of the project can be separated into four distinct areas:

- Chapters 1 and 2 introduce BOD and NetLogo, along with their positions in their relative fields. BOD from the point of view of artificial intelligence, in particular cognitive systems and robotics. NetLogo from the point of view of the field of agent based modelling and how it differs and compares to the key alternatives. By doing this I introduce the benefits of the proposed integration and identify the key strengths of both platforms that I need to preserve. This section is particularly useful to anybody who is unfamiliar with either BOD or NetLogo, but is also a useful introduction and perspective on the theory behind the two platforms and the proposed integration.
- Chapters 3 and 4 discuss in general terms the structure of BODNetLogo, as well as the rationale for the architecture I used and of any significant design decisions. This section is particularly of use to anybody who wishes to understand how I chose to integrate the platforms, whether to gain a better understanding of the program or to compare my approach to other possibilities.
- Chapters 5, 6 and 7 address the specific details and workings of BODNetLogo, as well as an introduction to the user interface and the use of the program. This section can be considered to be a very detailed guide to how the program actually works and why certain decisions were made and so is useful to anybody wishing to understand the implementation either to modify, analyse or simply use BODNetLogo to its fullest potential.
- Finally, chapters 8 and 9 analyse BODNetLogo, discussing the results of my approach, as well as how BODNetLogo meets its design aims and where it falls short, or needs some functionality not initially identified by the design. This section is useful to

anybody who is considering similar work and wishes to understand how well my approach worked, is considering furthering BODNetLogo itself or who wants a detailed appraisal of the final program.

In this chapter I discussed the basic ideas behind BOD and NetLogo, I also introduced my proposed program and specified the primary aims as well as the layout of this report.

# Chapter 2

# Background

In this chapter I will further discuss behaviour oriented design, its history, uses and position within the AI field. Similarly I will discuss the history and common uses of NetLogo as well as alternative modelling packages. By providing this background information I will introduce the key features within NetLogo that highlight its usefulness, as well as providing evidence to support my claim that BOD is a suitable option to increase its potential.

## 2.1 Behaviour Oriented Design

### 2.1.1 Reactivity and Behaviour Based AI

#### Reactivity

The focus of 'traditional AI' is on planning and search. In terms of an agent or a robot this involves the creation of a model of the world, as well as some search algorithm that can then formulate the best sequence of actions in order to reach a goal. However as Brooks (1991*b*) notes, progress had been limited, both in terms of ability, and in terms of performance. In particular he notes that the current state of the art was limited to a robot whose primary function was to pick up items (Lozano-Perez, Jones, Mazer and O'Donnell, 1989). Also that the search and planning approach led to extremely poor reactivity, with a plan cycle, and therefore each action, taking 15 minutes or more (Moravec, 1983).

The robotics community began to look away from the traditional AI approaches and formulate new ones. Brooks (1991*a*) produced several new robots that incorporated the idea of Behaviour Based AI (BBAI), which itself is also 'reactive'. Reactivity, or reactive planning attempts to avoid the problems with traditional AI by 'using the world as its own model'. This means that artificial agents such as robots no longer attempt to calculate plans based on their own model of the world, instead at every instant they take readings from their sensors and decide upon their next action, using some action selection mechanism.

**Behaviour Based AI**

Brooks (1991*a*) introduces BBAI, as the idea that human intelligence did not appear fully formed. Rather it was gradually produced through many years of evolutionary layering, with each new layer adding some ability. Intelligence is not therefore one module that deals with all aspects of thought and action, as traditional AI attempts to achieve through a 'perfect plan'. Rather it is a combination of simple behaviours each of which is capable of working independently or as part of an overall system to perform some function.

In his own robots Brooks implements this as a subsumption architecture. Where there are layers of behaviours, at the bottom for example there is obstacle avoidance; further up the stack there is wandering and exploring. Each layer is independent, and there is a hierarchy with lower down behaviours being able to override higher layers when necessary.

### 2.1.2 Action Selection

Action selection is vital for complex agents. Its role is to make the best choice of all possible externally expressed actions. This arbitration is necessary as externally expressed actions usually require exclusive control of an agent's physical state, for example a robot cannot travel in two directions at the same time (Blumberg, 1996). Non external actions or processes are not usually subject to action selection. In the case of complex agents this normally means choosing between two or more different options in a fair and consistent manner, which will satisfy the agent's goals.

Whilst Brooks' subsumption implements BBAI as a strict hierarchy there is no need for this limitation. In his aim to avoid any centralised control Brooks (1991*a*) does not explicitly define the action selection of his robots within his report. Instead he designs the architecture of the behaviours so that lower, less abstract behaviours such as collision avoidance can override other behaviours when necessary. It is possible to see that this hierarchy is in fact a form of behaviour arbitration or action selection.

### 2.1.3 Three Layer Architectures

Many of the ideas introduced by Brooks have developed and continue to dominate the field. In particular the concepts of behaviour based AI and reactivity are pervasive. In particular 'hybrid' or three-layer architectures are widely used, particularly in the robotics field (Gat, 1997). These three layer architectures share approximately the same structure:

1. *Skill layer*: stateless control loops which react in constant time to sensory information. These can be considered to be behaviours as in BBAI.

2. *Sequencing layer*: reactive plan execution controls the robot's actions based on pre-defined plans. Rather than having action selection inherent in the architecture, these

plans are separate from the functional code of the agent and allow the specification of complex, context sensitive action selection.

3. *Planning layer*: complex calculations and planning allow the benefits of traditional planning, search and learning techniques, whilst the sequencing layer preserves reactivity using existing plans. This functionality is not always required depending on the use and complexity of the agent, making this layer optional.

This architecture allows an optimal combination of reactivity and planning, whilst preserving the benefits of behaviour based AI.

### 2.1.4 BOD

Behaviour oriented design (Bryson, 2003*a*) has many similarities with three layer architectures, however it also has a number of key differences. Primarily, Bryson disagreed with the trend towards simpler and simpler behaviour modules arguing that there is no real intelligence without some state and that these simplified behaviours just passed the complexity to the planners. Consider the task specific nature of behaviours for example in a robot 'walking'. This is a functional behaviour or skill that can be debugged and tested in almost isolation and then combined with a full system. If the behaviour layer is restricted to simple non-state-holding functions then the plan must deal with some of the complexity from this behaviour and all other behaviours.

Bryson argues that it is far better to isolate all functionality and even learning related to a particular behaviour module. This then reduces the plan's complexity vastly, as action selection now only has to arbitrate between different behaviours when they want to express some external action. A useful side-effect of this modularity is that once the library of behaviours has been created, it can be used as many times as needed to design the intelligence of a particular agent, or other agents on the same platform, an example of this can be seen in section 2.1.6.

In addition to this new architectural layout, Bryson also proposes that the complexity of intelligent agents is limited by the complexity of the software design process (Bryson and Stein, 2001; Bryson and McGonigle, 1998). So BOD is also a design methodology. Due to the modular layout of the BOD's behaviour structures it is possible to take many of the best practises associated with object oriented design and reuse them, especially as the behaviours for BOD are traditionally written in an OOD programming language (Bryson, 2003*a*) with behaviours performing a role similar to a traditional object or class.

### 2.1.5 POSH Action Selection

As already discussed behaviours need some kind of arbitration to control their external outputs. In BOD this is done by Parallel-Rooted, Ordered, Slip-Stack, Hierarchical (POSH) plans (Bryson and Stein, 2001). The primitives of these plans are known as 'actions' and

'senses' which are interfaces to the behaviours and thus are the way a plan can control the actions of an agent. The primitives are also required to produce an instant or near instant response. This allows complex calculations and other functions to be happening in various different behaviour modules whilst the agent maintains is its reactivity.

### 2.1.6 Uses of BOD

Although it was originally designed for use within the robotics field, BOD has found a strong following in games and other virtual environments (Gaudl, Davies and Bryson, 2013). One of the primary platforms that BOD has been implemented for (Kwong, 2003), is the game Unreal Tournament. A good example of the complexity and variety of applications that BOD can be used for, the project allows users to create their own agent which can play the game, simply by creating an action plan, all the complex actions and senses which require communication with the game are held within the behaviour library. Gemrot, Brom, Bryson and Bída (2011) later used this implementation of BOD for Unreal Tournament to analyse the benefits BOD has over traditional unmodified programming languages. They concluded that BOD made it significantly easier to develop strong, complex AI behaviours.

Other examples include the generation of 'side quests' (Grey and Bryson, 2011) and within an alternative game genre (Gaudl et al., 2013).

## 2.2 NetLogo

### 2.2.1 Agent Based Modelling

Agent based simulation platforms (Railsback et al., 2006) are widely used in many different fields, including biology, sociology, and economics. They provide an efficient way to model complex situations. Due to this wide spread usage there are many different versions, each version having its own benefits, trade-offs and architecture style. Which platform is the best will therefore vary from situation to situation.

### 2.2.2 Platforms

Railsback et al note that one platform, NetLogo, has become popular due to its high-level nature, providing a relatively easy platform for academics with a background not associated with computer science. This obviously correlates to the previous assertion that many of the uses for these simulations come from different academic areas. The Railsback report also examines some of the other commonly used platforms, for example they note that MASON (Luke, Cioffi-Revilla, Panait, Sullivan and Balan, 2005) has a primary focus on being a minimal platform that will be extensively tailored and optimised for each specific use, providing a more efficient simulation. This means however, that there is a much greater overhead in learning and using the platform.

There is not a large amount of work available that compares the various platforms from a purely technical point of view, which means that much of the information it is possible to find in the literature can be found as part of the work of modellers who discuss their search for the best platform. Whilst it is possible to consider this domain-specific research less valuable, that may in fact be a mistake. For example a report that compares using general purpose platforms alongside industry specific competitors (Zhou, Chan and Chow, 2007) makes some good conclusions that provide a user's perspective. This users perspective is invaluable when forming a solution for the same users, such as the aim of this project. Specifically they conclude that SWARM (Minar, Burkhart, Langton and Askenazi, 1996) is a strong platform but has a major issue with complexity and therefore has a high barrier of entry, requiring strong programming skills and experience to use the platform efficiently. They also make a similar review of both MASON and Repast (North, Collier and Vos, 2006). Finally they turn to NetLogo and conclude that it provides by far the best non-programmer experience.

One paper, performing a review of NetLogo (Sklar, 2007) concludes that the platform's ease of use allows it to be used by a very large audience, much wider than its competitors. In particular the simplicity of creating a simulation, and a suitable interface is a large benefit, as they can be created using a drag and drop interface along with intuitive well documented menus. This point is reinforced by the large number of often influential papers published each year, and in widely differing fields, that make some use of NetLogo (Almeida, Kokkinogenis and Rossetti, 2012; Martin, Zimmer, Grimm and Jager, 2012; Vinatier, Lescourret, Duyck and Tixier, 2012; Whitehouse, Kahn, Hochberg and Bryson, 2012).

### 2.2.3 Action Selection in ABM

In a discussion of action selection methods in agent based modelling, Bryson (2003*b*) proposes that it is important to properly organise the intelligence of an agent when creating an agent based model. Specifically she proposes that modellers need to understand and identify which method of action selection they are using. Knowing this, the paper then discusses four key action selection methods that vary in complexity but also in capabilities. These methods include environmental determinism, finite state machines, basic reactive plans, and POSH plans.

NetLogo itself does not implement or enforce any particular action selection mechanism, instead it provides a programming language that, in theory at least, could implement any technique as needed by the author. However due to the simplistic nature of many of the models, and the limited programming skills of many of the users, the most common solution, based on the included 'models library' is to simply not do any action selection (See fig. 2.1, instead having each agent express all possible actions (See fig. 2.2).

Whilst having no action selection mechanism is seemingly a valid method, there are significant arguments against it, firstly because it greatly reduces the complexity of an agent, secondly it may lead to performance issues if every agent is doing several different things on each turn. But perhaps most importantly is the issue of realism, in order for turn based

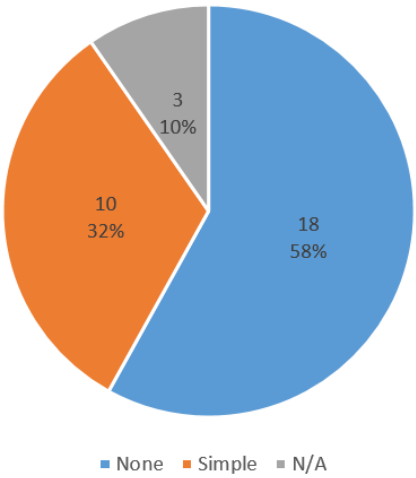


Figure 2.1: This chart shows the breakdown of the various action selection methods found in a typical sample of NetLogo models. In this case those found in the Biology sub section of the models library. None indicates that there is no action selection method, instead agents simply perform every possible action on each turn. Simple implies that there is a very simplified action selection method consisting of 2-5 if/else statements. A few of the results are N/A, this is for models in that library that use unusual methods or techniques that differ significantly from a traditional 'agent' based model.

actions to match in any way to reality there must be some realistic limit on how much an agent can do on each turn, the alternative as seen in fig. 2.2 is for a sheep to: move, eat some grass, and reproduce in a single instant.

Alternatively there is some basic use of if/else statements to implement one of the other very simple action selection methods.

```
ask sheep [
  move
  eat-grass
  death
  reproduce-sheep
]
```

Figure 2.2: This is the code that controls the sheep agents in the 'wolf sheep predation' model. Each agent is simply asked to perform each of the tasks available to it on each turn, this an example of a NetLogo model which uses no action selection.

Whilst the programming language is fully featured enough to implement a POSH action selection mechanism, the great complexity, combined with the lack of object oriented capabilities and the limit of a single script file makes the proposition of implementing a more

complex action selection mechanism within NetLogo unrealistic.

## 2.3 BOD MASON

MASON is a simulation platform similar to NetLogo that allows the creation of complex simulations. It varies from NetLogo in that it has a particular focus on extremely large simulations that may even be run on supercomputers (Luke, 2011), as such it is written and architected in such a way to allow the user a large amount of control at the cost of an increase in complexity, whereas NetLogo has a focus largely on ease of use, which makes it accessible to a wide variety of uses and users.

A previous project (Bryson, Caulfield and Drugowitsch, 2006) worked with success to integrate the MASON and BOD platforms, allowing a POSH action scheduler to control a MASON simulation.

Whilst MASON has many differences to NetLogo the overall aims of the project were similar. So many of the issues and solutions found by the authors will have an effect on my own work.

### 2.3.1 Turn Based Action Selection

One notable issue raised within this project is the conflict between 'an action-selection system designed for more conventionally humanoid AI such as robotics...' (BOD) and the more 'cycle-based ABM simulation platform' (MASON). This issue is important because it has the potential to produce incorrect results if there are unexpected interactions. It is easy to see the deep mismatch between many agents acting in real time and to their own schedule, for example people in the world react to stimulus at different rates, whereas cycle based platforms such as MASON or NetLogo repeatedly cycle through each agent, giving them a turn in some pre-defined order.

However the BOD MASON project found that a continuously iterating turn based system which goes to each agent in turn is actually a very good model of real time actions. This means that although there is a conceptual difference, in practice there is no real conflict.

## 2.4 Summary

In this chapter I discussed briefly the history of new AI which led to the creation of behaviour oriented design. This shows the position of BOD within the field of creating complete complex agents and the advantages it has over other methods.

Similarly I introduced the idea of agent based modelling and discussed the NetLogo platform, including the benefits that led to its popularity. I introduced one of the major

problems when using NetLogo to create complex agents, the difficulties in creating a complex action selection system. This is one of the rationales of this project. But in addition to the improved action selection of POSH, my aim is to bring the entire BOD methodology to NetLogo.

The other major advantage that BOD brings is the design process and principles. As NetLogo simulations are specified with no enforced structure in a single file, consider the possible complexity of a simulation that may only need basic action selection, but contains many different species, actions, senses and other functions.

It is the aim of this project to show that behaviour oriented design can both increase the capabilities of NetLogo in creating complex agents and at the same time increase the structure, modularity and design of the behaviour code.

# Chapter 3

# Objectives

In this chapter I will discuss the primary aims of this project, as well as what features and capabilities the final program, BODNetLogo, should include.

## 3.1 The General Structure

The primary reason for this project is to extend the functionality of NetLogo by allowing the specification of more complex agents. The reason NetLogo is preferable in this case to its competitors is because of its popularity. So it is vital that the design of BODNetLogo preserves as much as possible of what makes NetLogo so popular, its usability. In order to do this I replace functionality only when strictly necessary. Fortunately NetLogo makes available a well-developed API which gives access to the majority of its functions to outside programs.

The two primary components of BOD, are the behaviour library and the action selection mechanism.

### 3.1.1 The Behaviour Library

In a traditional NetLogo simulation this code is written in the customised programming language which provides access to all the functionality available to agents. There is only one clear issue with continuing to utilise the same approach in my own solution and that is that BOD behaviours are traditionally written in an OOD programming language (Bryson, 2003*a*) with each behaviour having its own object.

This allows different behaviours to run simultaneously to each other and the overall execution of the plan, allowing complex functions and learning to take place. However NetLogo code is written in a single script with no class or object notation. Functionally this will limit the potential of the agents and reduce some of the benefits of modularity. But it is important to consider that whilst some of the more advanced possibilities are removed by

this approach, the majority of BOD is still available, making a significant improvement over NetLogo as it is currently.

It is worth noting at this stage that due to the use-case specific nature of behaviour libraries, I do not need to do any further work with the behaviour libraries because any users will need to produce their own. But it is important in order to demonstrate and test the system that I produce a relatively simple complete simulation including a behaviour library. Also in my own example of an integrated agent I show how the design principles can be utilised (See section 7.3).

### 3.1.2 Action Selection

For the action selection mechanism however, there is no option other than to replace NetLogo's current control loop, which cycles through all the agents on each turn, allowing them to perform one action. This is because of the differences in NetLogo's current system and the complexities of POSH. Adopting my own control loop has a number of other implications which I will discuss further in Chapters 4 and 5. Finally, the last major component to address in this new system is the scheduler, which interprets the POSH plans when asked by the control loop and returns some action for the agent. So the general structure of BODNetLogo can be summarised as:

- A POSH Scheduler which interprets plans and produces the correct actions for the agents.
- A control loop which manages the simulation and agents.
- An interface with NetLogo to work with the user's behaviour libraries and other important NetLogo functionality.

## 3.2 Key Features

In addition to the main aim of integration for the project, BODNetLogo needs to have a number of other key features in order to be truly useful. As previously mentioned, ease of use is a big factor for the audience of NetLogo which is primarily educational and academics from a non-computer science background. So it is important that BODNetLogo does not add any complexity, that is not completely necessary, to the modelling process.

Secondly, BOD brings a number of key advantages both as an agent architecture and as a design methodology. It is key that BODNetLogo implements as many of those advantages as is possible, from the powerful action selection mechanism, to a strong example of how to implement behaviour libraries within NetLogo itself.

The final key feature is that the new system is capable of providing a suitable platform for scientific experiments, with repeatable, reliable results. It has been shown that both

NetLogo and BOD are capable of this as individual components, the task is therefore to ensure that my own implementation is true to their original design without introducing any new inaccuracies.

## 3.3 Design Requirements

In order to allow a good analysis of the final program it is important to consider and understand what the high level requirements of the final program are. These can be broken down as follows, with functional requirements indicating what is strictly necessary to consider the program as 'working' and non-functional requirements indicating features that whilst not strictly necessary would be of use to the users.

### 3.3.1 Functional Requirements

1. Allow a POSH action plan to control an agent within a NetLogo simulation.
   1.1. Run a POSH action plan using an implementation of an action scheduler,
   1.2. Allow the action scheduler to control a NetLogo agent using the API.
2. Allow Multiple Agents to be controlled within a NetLogo simulation.
   2.1. Allow duplicates of a single POSH plan to control different agents,
   2.2. Allow multiple POSH plans to control different agents.

### 3.3.2 Non-Functional Requirements

1. The application will be easily accessible to non-computer science users,
2. An example/template simulation is provided to demonstrate the correct use of BOD.

## 3.4 Summary

In this chapter I discussed the high level objectives that I wish to meet with this project. I intentionally place a lot of emphasis on the importance of ease of use, which when considering the current audience of NetLogo is important. There are many other simulation platforms available with better performance characteristics, or more customisation opportunities, so in order to justify my choice of bringing BOD to NetLogo I need to preserve that which stands out in NetLogo already.

I also hinted that this will force some compromises in the complex environment of BOD, in particular the loss of behaviours operating in parallel, and therefore their ability to perform complex planning and learning mechanisms. Also the true modularity of BOD, whilst it is

possible to preserve some semblance of modularity in a single script it is not optimal. But there is a trade-off between complexity and ease of use in almost any solution, it is my task to make the most suitable choices.

# Chapter 4

# Design

In this chapter I will discuss the general design of BODNetLogo, the rationale behind the language I chose as well as any other major design decisions.

## 4.1 Architecture

Due to the nature of this program, that is as an integration between two existing platforms, my aim is to preserve compatibility with future developments by keeping the design as modular as possible. This should allow, for example, an update to NetLogo to be integrated with little or no modification to the rest of the program. As discussed in Chapter 3, the basic functionality of BODNetLogo can be summarised into a few key modules. I have already introduced a few of these modules but I will now further discuss them and their basic functionality, the overall layout can be seen in fig. 4.1.

This is not a detailed breakdown of the individual classes or their functionality, instead it is a highly generalised representation of the basic structure of the program. For more in depth details please see chapter 5.

- Simulation Manager: This module is responsible for setting up the simulation, either from some configuration file, or from direct user input. It will collect details such as number and type of agents in the simulation, the NetLogo file to be used and any other details. It will then initiate the NetLogo platform using the API, and then create the Agent Manager passing along any necessary information.
- Agent Manager: Once created by the Simulation Manager module the Agent Manager will load all the specified agents along with their plans. After the agents are loaded the agent manager will create a control loop, iterating over all the agents and on each turn asking each agent to *fire* their action plan once.

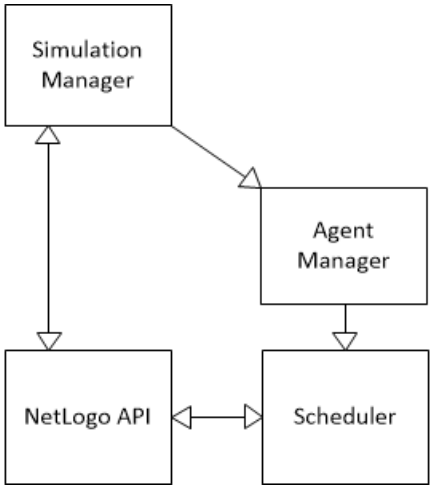


Figure 4.1: The high level architecture of BODNetLogo, it shows the major interactions between the various modules of BODNetLogo, with the arrows showing the directional flow of information through the program.

- Scheduler: Each agent currently running has its own action plan, and on each iteration of the control loop the plan is asked to *fire* this will result in the scheduler running the plan for the agent to find the action that should be produced. Once the correct action has been identified it will call it via the NetLogo API.
- NetLogo API: This module represents the interface with NetLogo, commands are passed by calling API methods.

## 4.2 Scheduler

The key decision to make when considering the POSH scheduler is whether to use and adapt an existing one or to implement a new one. It is worth noting at this point that there are many current implementations, the following are the ones which have publicly available and freely modifiable code:

- Lisp: This is the original implementation of a POSH scheduler by Joanna Bryson.
- Python: This is the implementation of a POSH scheduler written in python, however it has since been deprecated and replaced by:
- Python/Jython: Based on the original Python version, and modified by many different authors this is currently the most up to date scheduler, written largely in Jython, a version of Python which runs on the Java Virtual Machine.

- Mid-way through this project a further implementation in C# was completed and made available by Swen Gaudl. I will return to the discussion of this scheduler in section 5.1.3.

The currently recommended and most up-to-date version of the scheduler is the Python/Jython version. So, the decision to make was whether to use this version or to create a new one of my own.

There are several advantages and disadvantages to both options that influenced my decision. Firstly the option to simply re-use the existing scheduler:

- Reduces the amount of programming needed for the project.
- Does not require any extra correctness testing as it is already known that the scheduler works properly.
- Many people have already worked with the scheduler, re-use would allow them to carry their experience and knowledge over.

However:

- The use of NetLogo to implement the behaviour libraries rules out much of the additional functionality available in BOD, this means that a significant portion of the scheduler would be redundant.
- Whilst Jython is useful to access Java functionality from Python, the opposite, accessing Jython functionality from a Java program is limited and complex. This could cause unnecessary inefficiencies in any interactions between NetLogo and BODNetLogo, or even limit functionality, as NetLogo is written largely in Java.

The advantages and disadvantages of creating a new scheduler in Java are largely the converse of the above reasons. Based on this information and after looking at the possibilities of each choice in detail I found that the best option is to create a new version of the scheduler in Java.

The most significant rationale for this was looking at the additional complexity and the amount of alteration to the base scheduler that was needed for example to work with Unreal Tournament, or MASON (Kwong, 2003). Whilst causing some extra work for this project I believe that a native scheduler would significantly reduce problems for users.

Traditional schedulers maintain a broad level of generality (Kwong, 2003; Bryson, Caulfield and Drugowitsch, 2005). They are a platform that in principle can be modified to work with any application, however as already noted NetLogo is not able to access some of the advanced functionality of BOD, and also this implementation is unique in that the behaviour library will be implemented inside of NetLogo. This means that creating a generalised scheduler would in fact be extra complexity. Considering this, my own scheduler

need not be as broad as other implementations. Instead a more tailored approach will give better results for BODNetLogo. More specific details can be found in section 5.1.3.

Despite my approach in not creating a general scheduler, there will still be a significant portion of the work that can be used to form the foundations of a more generalised version at a later time. As there will not be any major component removed, nor any significant design changes, the work to generalise the scheduler at a later time whilst not trivial should not be unsurmountable.

## 4.3 Programming Language

With the decision to create a new scheduler in the Java language, along with the fact that NetLogo itself is written partially in Java as well and thus proves a powerful API (Application Programming Interface) in Java. It becomes clear that the best option for my own code is in Java, having only one language for the entire program will preserve the maximum amount of functionality, as well as simplifying the design and implementation. Additionally using only Java will maintain the current platform independence that NetLogo already has, allowing access to users on any operating system.

However it is important to consider any problems that might arise from using Java, by doing this it may be possible to minimise or at least predict any future issues.

The most significant of these problems is likely to be performance, due to its platform independent architecture Java is known to be slower in large, complex calculations than some other alternatives. This is because Java is run in a virtual machine rather than being compiled directly to an executable and also because it does not allow fine grained control of memory and other system functions. However it is worth considering that NetLogo is already considered to be slow compared to its competitors (Sklar, 2007) and that its users therefore have made already sacrificed raw speed for functionality and ease of use. Whilst this does not imply that an inefficient scheduler is acceptable it does at least justify making any performance issues a lower priority.

## 4.4 User Interface

As NetLogo already provides a strong Graphical User Interface (GUI), the need for an interface for BODNetLogo is limited to a small subset of activities. In particular the two main elements of the user interface for BODNetLogo are firstly to collect relevant information needed to run the simulation, for example the location of the NetLogo simulation file, and the location of the '.lap' (learnable action pattern) files which contain the dynamic POSH plans for the agents. Secondly BODNetLogo needs to implement a new interface to control the simulation, including play/pause, run for a certain number of iterations and any other useful options.

Text based UIs are quicker to develop due to the simplicity of recognising a few typed key words against that of designing a robust *frame* of graphical elements with their reporting functions and all other issues commonly associated with GUIs. However GUIs are much more discoverable and more intuitive and as NetLogo has a large audience in the educational and other academic fields it is likely that in the main users of BODNetLogo will be unfamiliar with text based, or command line, interfaces (CLI).

Considering this BODNetLogo implements a basic GUI which collects the necessary simulation details and then allows control of the model. Additionally I have implemented a basic configuration file reader, this reader allows users who are running the same simulation repeatedly to skip specifying their simulation in full on each run by writing a simple configuration file which is then read by BODNetLogo and the simulation is set up based on these options.

This configuration file functionality is only implemented to a basic level in the current version of BODNetLogo such that it is only capable of reading the files, there is not yet any mechanism to save or create them, instead they must be written by the user. Whilst this functionality will be added at a later point, what is implemented already is not intended to be a general solution, instead it is a useful tool for frequent users and as it is relatively simple to create such a configuration file the urgency to enhance the functionality is limited.

## 4.5 Summary

In this chapter I discussed the general design elements of BODNetLogo in order to provide a broad overview of the structure and functionality, more detailed information can be found in chapter 5. The general structure of BODNetLogo has been described, including the general layout of the various modules. The modular design is particularly suitable for this project as an integration between two different platforms as NetLogo is widely used platform that often receives updates and patches. Similarly BOD and in particular the POSH action scheduler is often updated with new functionality. The design of BODNetLogo allows these changes to happen without breaking the integration and allowing a wide compatibility range.

I have also described the reasons for choosing the Java language, the key advantage is the use of a single programming language which allows the maximum possible freedom in design and functionality as different parts of the program are not limited by being forced to work through some interface.

Finally I have described the high level design of the user interface with the BODNetLogo program, setting out the two areas that need a new UI, the simulation set up and the simulation controller. These are relatively simple tasks that can be performed via an easy to use and intuitive graphical interface that should not significantly compromise the usability of NetLogo.

# Chapter 5

# Implementation

In this chapter I will discuss the important details from the implementation of BODNetLogo, including the general class structure, some of the important classes and how they interact. This is to give an understanding of the implementation and workings of BODNetLogo.

## 5.1 Class Structure

Figure 5.1 shows the class structure of BODNetLogo, along with arrows to represent significant interactions between classes. Each section will be described in detail in order to explain how BODNetLogo functions.

Considering the architecture as a whole it is possible to see the relation to the architecture principles set out in chapter 4, with the modularity between the key functionality which is largely focused in the Simulation Manager and Agent Manager modules. This means that the scheduler is largely interchangeable should that be required, or more likely incremental improvements can be made to the scheduler without affecting BODNetLogo as a whole.

Similarly, although not shown on the diagram the modularity which helps to insulate from changes to NetLogo is present. For further details see their respective sections but as a brief explanation the NetLogo API is used in two key areas of BODNetLogo, firstly in the simulation set up which co-ordinates the details of the simulation with the NetLogo plan and then in the POSH primitives. So, should the NetLogo API be changed in any significant way there is only minor exposure that would need to be corrected.

The GUI structure is not represented in this diagram as it not a fundamental part in the structure of the program. However it is still important to consider, so I discuss the implementation and class structure of the GUI in section 5.1.4.

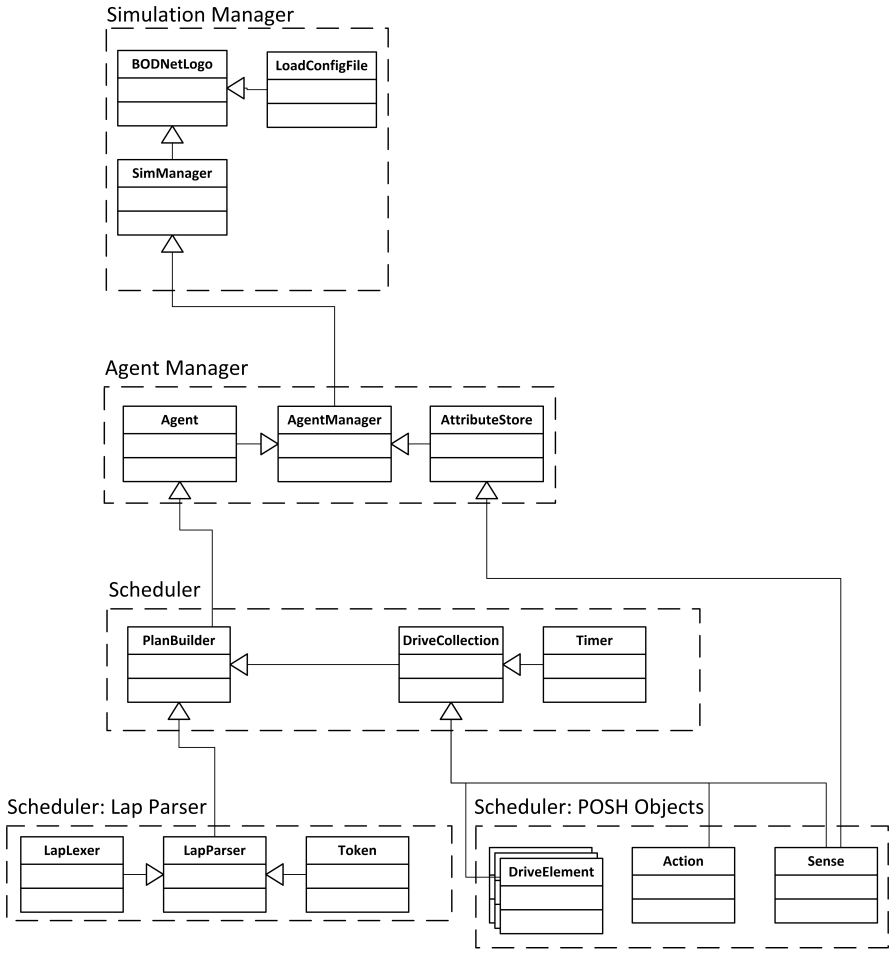


Figure 5.1: The detailed class structure of BODNetLogo. The arrows represent the flow of information and in general the flow of the order of execution in a typical simulation. The stacked elements in POSH Objects represent several classes that make up the elements of the drive collection. Action and Sense are highlighted because they are the most significant elements of that module.

### 5.1.1 Simulation Manager

The simulation manager module of BODNetLogo controls the creation and initiation of the overall simulation before passing control to the agent manager which handles the agents and the running of the model.

#### BODNetLogo

This is the location of the 'main' function that controls the start of BODNetLogo, specifically it creates the first GUI where the user is asked to input the configuration details, which include the following details, also note that further details on the GUI can be found in chapter 6:

- Location of the NetLogo simulation file which has already been created with the behaviour library and any other details for the environment or reporting tools. The number of agents, any predefined variable values, and other agent characteristics should be defined in a 'setup' function within NetLogo.
- Specification of each of the breeds, the quantity of agents for each specified breed is later collected from the NetLogo simulation this is so that it is not necessary to specify quantity twice and risk unnecessary errors. The specification includes:
  - Breed name as defined in NetLogo, both singular (e.g. wolf) and plural (e.g. wolves).
  - Location of the lap file.
  - Breed attributes, more details on attributes can be found in section 5.1.2 in the discussion of the attribute store.
- Any other configuration options, for example the output from the model the user requires.

Once the user has entered the required information BODNetLogo starts NetLogo via the API and then passes control to the simulation manager class.

#### SimManager

SimManager continues the execution of the program from BODNetLogo by interacting with the NetLogo model which is now open after being started by BODNetLogo, for example it will run the 'setup' command that is present in NetLogo models, as well as collecting information about the quantity and type of breeds that are present in the model.

Once the simulation has been set up and any information gathered from it, SimManager will create an instance of AgentManager and pass all the necessary information in a data structure.

### 5.1.2 Agent Manager

#### Agent Manager

Using the data from SimManager, AgentManager initialises the AttributeStore and then creates an instance of Agent for every agent active in the simulation. Each new Agent instance is given an ID which corresponds to their value in NetLogo, as well as a *String* value which contains the information from the lap file which was specified for that breed. As part of the Agent instantiation the plan *String* is interpreted and each Agent thus gains a DriveCollection, which is the container for POSH plans.

The control loop then cycles through all of the Agents in turn, asking their DriveCollection to fire once. This has the effect of each agent acting once in a turn based fashion. This simple control loop is sufficient for a basic model however NetLogo agents have the capability to both 'spawn' new agents and to 'kill' other already existing agents. Consider this in terms of the *wolf sheep* example already introduced, this is the same as giving birth or a wolf eating a sheep. It is important to manage these two things because for each agent to act in turn BODNetLogo must maintain an accurate store of instances of Agent that correspond exactly to the agents that are active in NetLogo.

Thus the control loop also checks for new spawns on every cycle, this is done by asking NetLogo for the number of active agents, if there are any new agents then BODNetLogo finds their species and creates an instance of Agent to control the new spawns. For deaths the control loop simply monitors the outcome of the Drive Collection, if the agent is no longer active then NetLogo returns an error when it is asked to perform some action, this is handled by BODNetLogo by removing the corresponding Agent from the agent store.

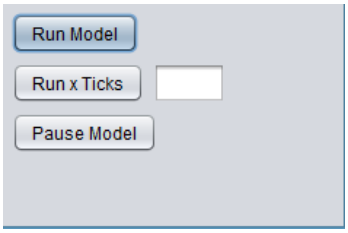


Figure 5.2: The BODNetLogo model controller which interfaces with the control loop.*Run* and *Run x Ticks* start the control loop, which then iterates through the list of active agents. *Pause*, pauses the control loop at the end of the current full iteration so that each agent has always had the same number of turns.

The control loop is managed by the model controller graphical interface (See fig. 5.2) which allows the model to be paused or run, as well as run for a specified number of 'ticks' or iterations. Currently the control loop only responds to the 'pause' command at the end of every iteration, this may not be ideal because in some models particularly those with a

large number of agents each iteration can take a relatively long time, this makes the model appear unresponsive. On the other hand it could be argued that when attempting to model real time scenario in a turn based manner it is only fair to consider results that appear at the end of each full iteration.

The control loop is also responsible for running an optional *nature* function at the end of every turn. This is a tool for users to extend the capabilities of their model to include some *observer* functionality. This could be something related to nature as the name suggests such as controlling the patches that make up the world, or it could be used to simplify the complexity of calculations. For example if there is some flock of agents that needs to calculate the centre of the group, instead of every agent computing this value it is logical for it to be calculated once at the end of each tick. To utilise this feature the user simply needs to create a function called 'nature' inside their NetLogo model (See chapter 7). If a model does not implement this function the control loop just skips over it.

#### AttributeStore

Senses involve the return of some value from the behaviour so that the scheduler knows which path of the plan to follow. NetLogo has *functions* which is how actions and other behaviour related functionality is encoded, the API allows these functions to be called simply by sending the name of the function. However there is no simple way to have the API return any output from these functions to the calling program, so it was necessary to implement some other method to allow senses to communicate the value. NetLogo agents have 'local variables'. Local, in that every unique agent has its own instance of the breed specified variables with their own value. For example if sheep are specified to have an energy variable every agent of the breed sheep will have its own value for energy that will likely fluctuate over time.

The NetLogo API provides a method to access these variables so this is how senses are implemented. Unfortunately the API only allows access to these variables via an index value and not by their name, so to continue the example of sheep agents, if the plan has a sense that checks energy levels it is not possible to directly ask for the value of energy. Instead the index value for energy in the array of all the variables that is possessed by the sheep breed must be requested.

In order to deal with this BODNetLogo implements the AttributeStore which records all of the breeds in the current model and all of their attributes or 'local variables' along with the index values. So a sense with name *checkEnergy* asks the attribute store for the index value of energy for the sheep breed, then using the index value to ask NetLogo for the value of energy for the current agent. This approach is functional but has a number of drawbacks, firstly that it creates additional work for the user in specifying their BODNetLogo model. More importantly however it is a source of potential errors because it is reliant on users to enter the attributes for each breed in the exact same order they are specified in NetLogo so that the correct index values can be calculated.

### 5.1.3 Scheduler

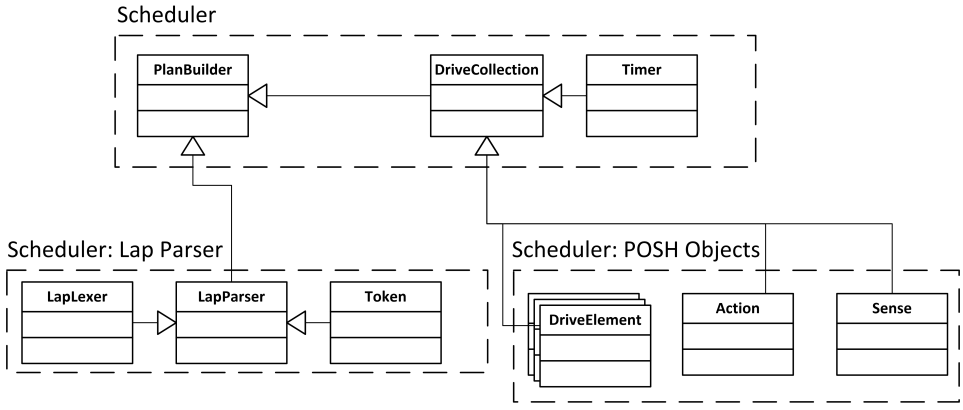


Figure 5.3: The structure of the POSH scheduler. The arrows show the flow of information between the elements, the PlanBuilder class is passed a String containing the POSH plan by the AgentManager, it then constructs and returns an instance of DriveCollection which is made up of POSHObjects.

As I have already discussed, the best option for BODNetLogo is to implement a new tailored version of the POSH scheduler that is implemented in Java and removes any unusable or unnecessary features, both for simplicity and efficiency.

However rather than creating this program from scratch. My own implementation is a part translation of the existing implementations with some modifications. I worked partially with the Jython version as the most fully featured of the available schedulers and the one that is currently recommended, however the new implementation by Swen Gaudl in C# was the primary basis. C# has a very similar syntax and semantic structure to Java which means that some of the classes, particularly the more basic ones were direct copies with minimal modifications. However the larger classes as well as the NetLogo specific ones such as 'actions' and 'senses' still required a significant modification or even a complete rewrite.

I will provide more detail on the workings on the scheduler in the rest of this section, and will indicate the more significant work that took place to translate the scheduler.

#### Scheduler

Once an Agent is instantiated with a String containing the information from the given lap file it creates an instance of PlanBuilder, passing in the String. PlanBuilder will use the POSH plan to create a data structure in the form of a drive collection. A drive collection is the term applied to the root container of a POSH plan, with each drive element then being one of the other structures available to POSH planners, such as a competence or action pattern (Bryson and Stein, 2001).

The first step to create this data structure is to pass the String to the lap parser module which will return a data structure with all of the formatted information taken from the file. This data structure is then used by PlanBuilder to create an instance of DriveCollection which is made up of all the relevant instances of POSHObjects. Once this drive collection has been created it is returned to Agent which can then fire it to produce some action.

PlanBuilder is the largest and most complex part of the scheduler with the ability to create any possible combination of drive elements from a valid plan. It is largely the same as the C# version, however some necessary and fundamental changes had to be made to the data structures used and returned by the lap parser. This meant that a very large number of minor changes were necessary.

Timer is one example where it is possible to see BOD's root in robotics. It is used so that elements of a POSH plan can have some maximum frequency in when they occur. This helps to balance dynamic action selection with the relatively slow speed of physical motion, such as the movement of an arm or rotation of a wheel. However it is still a useful feature in virtual reality and agent based modelling to give more realistic behaviours and provide a way for a plan to contain high priority but low frequency actions. Consider for example the sheep model I have already discussed. An action plan for a sheep may contain some part that checks if a wolf is nearby, this is obviously a higher priority than eating or wandering, however without the capability to specify frequency the sheep will be constantly checking for a nearby wolf instead of eating or any other lower priority action.

The major change that was required to timer from its current real-time implementation was to change it from a clock which allowed action frequency to be specified in terms of seconds or minutes to a turn counter that instead measures *ticks*, the name NetLogo gives to an iteration through every agent.

### Scheduler: Lap Parser

The Lap Parser module takes a String containing a POSH plan and returns a data structure with all of the information necessary to create a drive collection. The parser works through lexical analysis, working through the plan to find all the required information and creating a *Token* for each segment. This allows the program to find incorrect syntax in the plan and, assuming no errors are found, enables LAPParser to create a large *Tuple* data structure.

All three of the class files are largely based on the implementation in C# with no significant design differences. The only major changes made were due to a difference in the capabilities of data structures provided by Java, such as ArrayLists, HashMaps and Dictionaries. These differences mean that the majority of the methods in the classes have a small differences compared to previous implementations but the overall structure and functionality is the same.

### Scheduler: POSH Objects

The POSHObjects module contains a large number of elements that represent parts of a POSH plan, including the directly comparable examples such as *Competence*, *Action pattern*, or *Action*, and also the classes that are not specific elements but instead are necessary to provide functionality. Examples of these include *Trigger*, *FireResult* or *ElementCollection*.

These objects are all used to by a drive collection to store the POSH plan in a structure that can be fired to produce some action based on the values returned by the senses of an agent. Each fire produces only one action, allowing for a fair turn based scheduling. As when each agent can act in turn the model can accurately represent a real-time system where agents act simultaneously (Bryson et al., 2005).

Actions and senses are one of the few points of contact with NetLogo, as part of the modularity that maximises compatibility with future versions. Specifically actions call a NetLogo API function which sends a command that in turn calls some function in NetLogo for a specified agent. This is how the POSH plans can interface with the behaviour library.

As discussed in section 5.1.2 senses need to use an attribute store that gets the index value of a particular variable held by a NetLogo agent, this value should be a decimal number that is then compared to the sense condition given by the plan. However senses are not just limited to some decimal value, BODNetLogo also allows users to create a function that updates the sense value before it is taken.

For example a sheep sensing if a wolf is nearby cannot just know the answer to this, instead it needs to look. My implementation of sense allows for an optional function in NetLogo that is called prior to receiving the value. This optional function must be named 'update_x' where x is the name of the relevant sense variable. This gives two possible flows for a sense, continuing the sheep example:

- To sense how much energy is available:
  - The value of 'energy' is modified continuously by any action that uses or produces energy, e.g. walking or eating,
  - The amount of 'energy' can be retrieved at any time with no updating required by getting the correct index value from NetLogo.
- To sense if the sheep is in danger:
  - The value of 'danger' is not actively modified by any other action,
  - When a sheep needs to check if it is in danger it runs the sense 'danger', which:
    - Runs update_danger,
    - Retrieves the value of 'danger', which could be 0 for false or 1 for true.

In traditional, general POSH schedulers, *Action* and *Sense* do not interface directly with the platform they are operating on, instead they pass through a behaviour store which provides

a record of all of the behaviours and primitives available to the planner, these behaviours which are implemented in some OOD language are then called and produce some action or return some sense value. As BODNetLogo does not have a behaviour library written in an OOD language and NetLogo provides an API command to call any function directly there is no need for the additional complexity and layer of the behaviour store. Whilst this loss of generality does prevent my implementation of the scheduler being easily modified to work with a platform other than NetLogo, it does improve BODNetLogo as a whole.

Actions and senses are just one key area of modification in POSHObjects, although the objects are largely the same as previous implementations I have made changes where it was possible to simplify or improve the functionality.

### 5.1.4 GUI

Figure 5.4 shows the class structure of the components that make up the graphical interface. In this section I will discuss how these classes interact with the rest of BODNetLogo, for a more detailed discussion of the functionality and appearance of the GUI please see chapter 6.

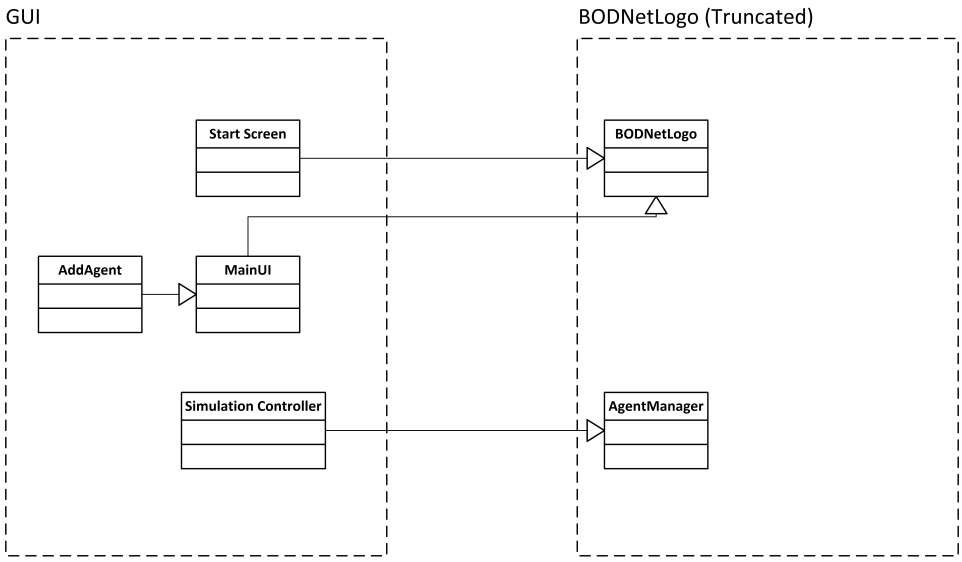


Figure 5.4: The class structure of the BODNetLogo graphical interface, showing which part of BODNetLogo the elements interact with, the first set of elements gather information about the model from the user. Once the simulation has been created, loaded, and the control loop started, SimulationController interacts with the loop.

The GUI can be broken into two key sections, firstly the larger part is that which collects all

the information needed from a user in order to run a BODNetLogo model, this functionality is performed by the classes: 'StartScreen', 'MainUI', and 'AddAgent'. Specifically start screen asks the user if they would like to create a new simulation or load a configuration file. Both options then trigger the MainUI, if a configuration file was selected the elements of MainUI are already populated, if no configuration file was selected then the fields are blank. Once these screens have been completed they pass their information to the main BODNetLogo class which then uses the information to create the model.

Simulation Controller is the other key section and is a simple screen that provides the user options to interface with the control loop, for example starting or pausing the model.

## 5.2 Summary

In this chapter I discussed in some detail how BODNetLogo was implemented as well as the rationale that supported any important decisions. I further emphasised the modularity of BODNetLogo that protects compatibility whilst allowing for new versions of NetLogo and upgrades to the POSH scheduler. The major component of this is that NetLogo is only accessed in two main areas, in the initial setup and in the primitives.

Additionally BODNetLogo only relies on the scheduler to return some drive element that is capable of being 'fired', it is therefore possible to make changes to the scheduler for efficiency or even for additional features by altering the firing process, as long as the output is some action there is no unexpected effects on BODNetLogo.

The attribute store is a good representative example of the compromises that needed to be made to satisfy the goals of this project. It requires some additional user input as a trade-off for allowing the behaviour library to be written in the simpler NetLogo language. It is also an example of the limitations of using an API, if NetLogo allowed the value of a variable to be retrieved based on its name rather than its index value then AttributeStore would not have been necessary. Another important idea that is discussed in this chapter is the reuse of the existing implementations of POSH schedulers. Rather than attempting to design a scheduler from scratch I took ideas from, and in the case of the C# version borrowed heavily from, the previous versions. This simplified the task and allowed me to focus on other issues that had not already been solved by others.

This is not to say that the translation was in anyway trivial, subtle differences in programming languages and the data structures that they provide meant that I had to consider the workings of each section of the code to ensure that elements were performing as they should. An example of this is the difference between a C# Dictionary and a Java Hash Map, on the surface they are very similar and certainly replacing a Dictionary with a Hash Map in a translation produces working code, however there is a significant difference. Dictionaries preserve the order of the data that was given to them, whereas Hash Maps do not.

At one point the various drive elements are stored in a Dictionary, translating this directly

to a Hash Map produces code that appears to work, however this completely destroys the order of the elements, which is very important because the order signifies the priority of the plan elements. This is just one example of the difficulties of translation, and although it was less work than creating a scheduler from scratch it was certainly not trivial.

# Chapter 6

# User Interface

In this chapter I will provide an introduction to the interface of the program, along with an explanation of the functionality of the major elements.

## 6.1 Start Screen

Figure 6.1 shows the initial screen that is presented to users upon the start of BODNetLogo. Along with a basic introductory text it asks the user if they want to 'Create New Simulation' or 'Load Existing Simulation Setup'. Both choices will take the user to the next screen, the MainUI. Choosing to create a new simulation will load a blank configuration with no elements already specified. Whereas choosing to load a configuration file will present a 'File Chooser' menu where the user can specify a configuration script. BODNetLogo will parse this script and populate the elements of MainUI with the options specified in the configuration file. The user can then choose to simply start the simulation based on the configuration file or can add further information.

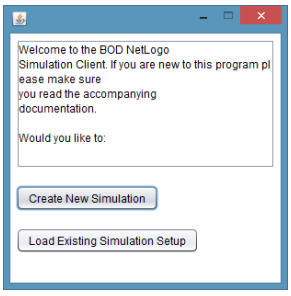


Figure 6.1: This is the first screen seen by users, if load a configuration file is selected, a file browser appears for the user to select their file. This information is then passed to BODNetLogo so that it can pre-populate the elements of MainUI

## 6.2 MainUI

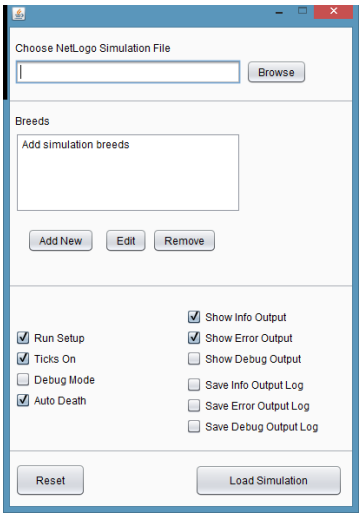


Figure 6.2: This is the screen where general simulation settings are gathered, this allows BODNetLogo to load the users simulation file and implement their preferences.

The MainUI (See fig. 6.2) is the screen that collects the general simulation details from the user. Specifically it collects the following:

- The location of the NetLogo script which contains the definition of the agents, the behaviour library and any other NetLogo elements. This is a normal '.nlogo' file.
- The Breeds that are in the simulation, see section 6.2.1 for more information on what details are collected about each breed. The white text box will show a list of all currently added breeds so that a user can select one of them and remove or edit it. If edit is chosen then the AddAgent screen is loaded with the details that have already been specified pre-populated.
- Simulation Options:
  - *Run Setup*: This option is true by default and tells BODNetLogo that it should run a function called 'setup' once the NetLogo simulation has been loaded. In traditional NetLogo models the setup command loads the agents and specifies the environment along with any other user added code. This should be left as true unless the user has a specific reason to un-check it.
  - *Ticks On*: This simply tells BODNetLogo to send a 'tick' command to NetLogo at the end of each turn, this allows NetLogo to keep track of the progression of 'time' in the simulation so that its reporters, particularly the graphical ones can keep a time axis to allow meaningful representation of any results. It is possible

that users may wish to use some other time metric and thus write their own tick command in their code, in this case *Ticks On* can be disabled.

- *Debug Mode*: This is a basic but useful feature that adds a time delay after every agent's turn so that their actions can be more easily followed in the model view.
- *Auto Death*: This tells BODNetLogo to send a *die* command to NetLogo once an agent meets the goal of its drive collection. For example if a drive collection is representing some living thing the 'goal' may be that the agent runs out of energy, once this happens they should die. If Auto Death is set to false when an agent reaches the end of its drive collection it will be removed from the control loop but will not be removed from the NetLogo model. This may be useful if a user wants to see where their agents meet their goal for example.

- The final set of options simply specify what text output they want to see from BODNetLogo. *Info* is general information about the model. *Errors* are exceptional problems that occur and *Debug* is a wide range of statements that show the progression through the control loop and each agents action selection.

### 6.2.1 AddAgent

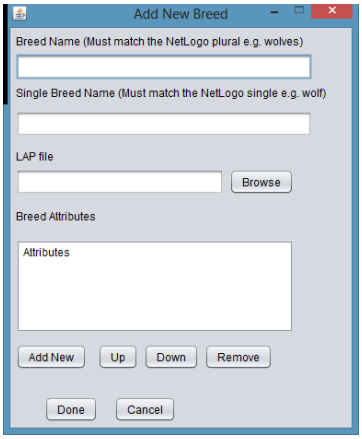


Figure 6.3: This is the GUI screen where users are asked to specify an individual breed that will be part of the BODNetLogo simulation. Choosing *Add New* loads the add attribute screen. The *Up/Down* and *Remove* arrows allow the user to control the already specified attributes.

The *Add New Breed* screen which can be seen in fig 6.3 collects the various pieces of information about the agents in the NetLogo simulation that BODNetLogo needs to run. The quantity of agents for each breed is not specified by the user here, instead BODNetLogo is able to get this information directly from NetLogo, this helps to prevent some repetition. AddAgent collects the following information:

- The name of the breed/species of agent. This should identical to the plural value given to NetLogo, for example ‘wolves’ or ‘sheep’.
- The name of a single agent in the breed/species. This should be identical to the singular name given to NetLogo, for example ‘wolf’ or ‘a-sheep’.
- The location of the lap file for this breed. This is the file that contains the POSH action plan, either written by the user or generated by ABODE.
- The breed attributes, this is a list of all the attributes possessed by the breed, whether the user intends to access them or not. The list must be in the same order as the attributes were specified in NetLogo, this is so that AttributeStore can properly calculate the index values. Clicking *Add New* brings up a pop up screen (See fig. 6.4). *Up* and *Down* reorder the selected item in the list, whilst *Remove* deletes the selected value.

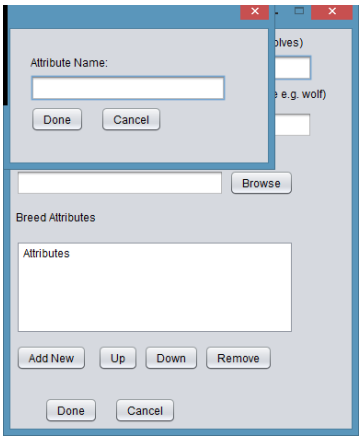


Figure 6.4: This pop up screen allows the user to input a new attribute for the current breed.

## 6.3 Simulation Controller

Figure 6.5 shows the simulation controller window that appears alongside the NetLogo screen when the model has been loaded and the plans generated for each agent. The user can tell the model to start/resume, run for a specified number of ticks, this means iterations as defined by BODNetLogo, with one tick representing each agent having one turn. If the user decides to disable BODNetLogo ticks and use their own timing system, running for x ticks will still refer to one iteration through each agent and not the new user specified ticks.

*Pause Model* tells the control loop to stop at the end of its current iteration, rather than stopping immediately. The rationale for this is explained in section 5.1.2.

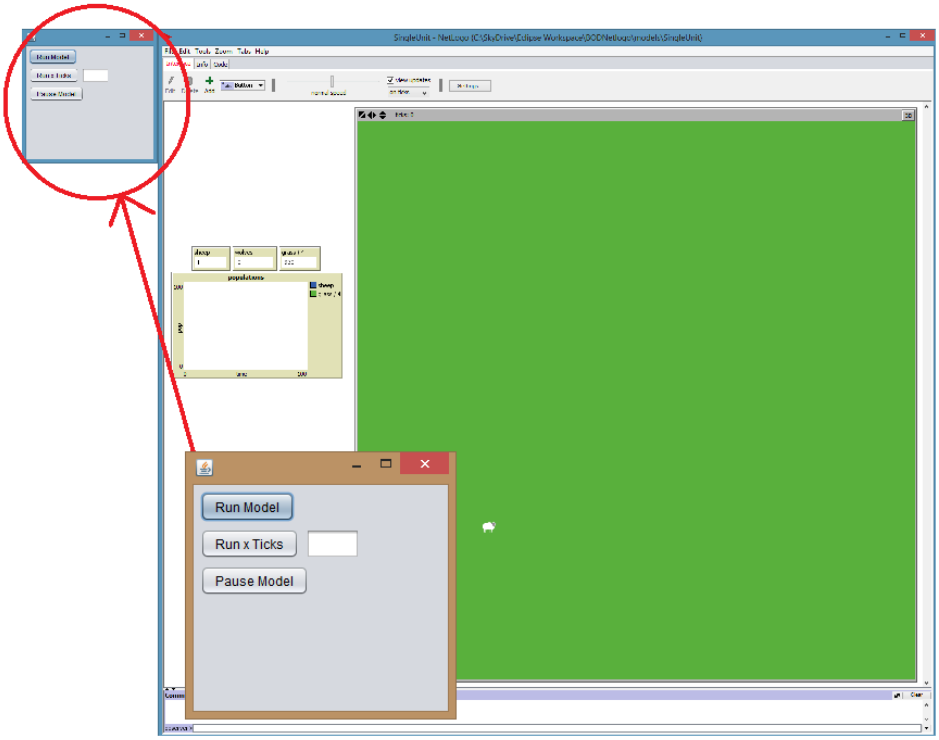


Figure 6.5: This screen shows the simulation controller as it appears alongside the normal NetLogo user interface, the controller interfaces with the BODNetLogo control loop.

## 6.4 Summary

In this chapter I discussed the Graphical User Interface of BODNetLogo. It is a fairly simple interface as NetLogo already provides an excellent and customisable interface for users to collect information about their model. The BODNetLogo interface therefore only needs to collect any extra information needed for the POSH action selection and in the case of the simulation controller provide a way to interface with the functionality that has been superseded by BODNetLogo.

The functionality behind the interface is also fairly simple at this stage. There is not a large amount of robustness in the code as it stands. This means that should a user enter incorrect values in one of the fields BODNetLogo does not immediately identify the mistake to the user, instead it will find the error at some later stage, stopping execution with a generic error code. Although this is not ideal for general use software it is safe to assume the audience of BODNetLogo will be fairly technical, meaning that robustness whilst preferred is not an immediate concern.

# Chapter 7

# Testing

In this chapter I will discuss some of the tests that I performed to ensure that the program meets the design aims for the project. As I said in chapter 4, both NetLogo and BOD are already proven platforms, thus it is not necessary for this project to validate or verify their correctness. Rather it is necessary to test that my implementation of the scheduler and other BODNetLogo components does not create some unexpected interference. This is done by creating a model that serves to test the correctness of the various elements as well as acting as an example to future users. In addition to this single plan I also performed a great number of individual tests, testing different possible combinations of plan elements and varying levels of input into BODNetLogo.

The aim of this example model is not to be very complex or even to have any real value as a simulation. Instead it should simply be something that can be a good demonstration and at the same time tests as much functionality as possible. To that end I reproduce the 'Wolf Sheep Predation' model that is used by NetLogo to teach new users the platform. My own version was created using the design principles and heuristics described by Bryson (2003*a*) however for the sake of brevity I will only discuss the current state of the model and not the iterations that BOD follows. The components of the model can be found in Appendix A.

## 7.1 Setup and Configuration

As part of this example I also implemented several of the optional features in BODNetLogo in order to test and show of the more in depth functionality. This included:

- A *Setup* function inside NetLogo which specifies the number of each agent to create at the beginning of the model, as well as their shapes in the visual viewer, and their starting attribute values. In addition to the agent specification, setup also defines any global values, and any other code needed for the model for example values for NetLogo *monitors*.

- This model also implements the optional nature feature. Specifically nature controls the growing of grass at the end of each tick, but it could also be used to allow complex calculations needed by agents to be performed once each turn rather than by every agent. This feature is utilised by creating a function called nature that is then called by the control loop at the end of each tick.
- The details needed for BODNetLogo are stored in a configuration file which can be loaded at the start of the program to save the user having to repeatedly enter the same details each time the model is run.

## 7.2 Behaviour Library

The behaviour library represents a potential source of both weakness when compared to other BOD implementations and strength when compared to other NetLogo models. That is because it is no longer possible to utilise a fully OOD language to build the library, instead it must be constructed in a single script file within NetLogo. However the design principle of modularity can be carried to NetLogo in order to reduce the scope for complexity in large models.

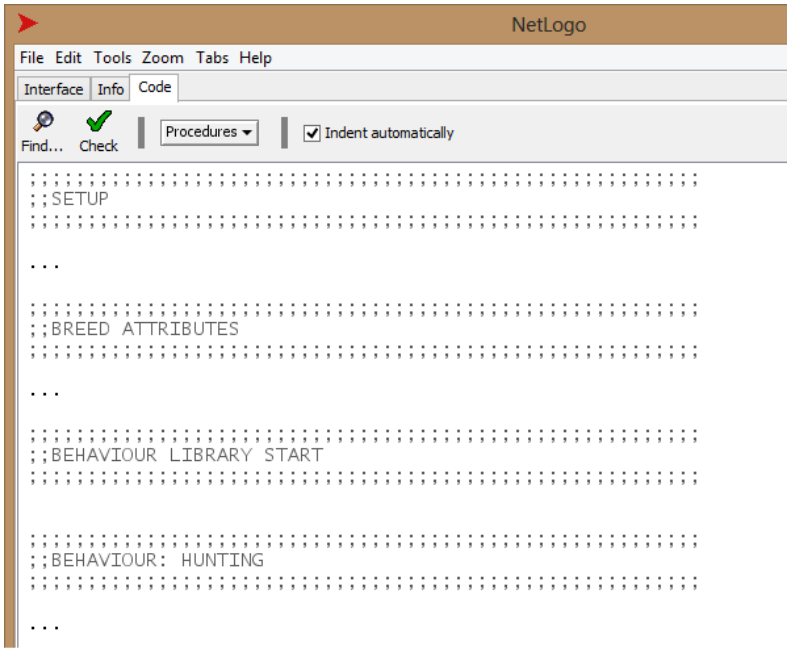


Figure 7.1: This is an arbitrary standard I have created to allow the behaviour library to remain modular. Although the code is actually all in the same script, the sectioning creates an impression of modularity for the user.

Although my own choice of Behaviour 'module' demarcation is fairly arbitrary it provides a good representative example of the principle and allows a large simulation to be created without complicated code. (See fig. 7.1).

The behaviour library for this particular model is relatively simple, consisting of four different behaviours:

- Grazing: Provides the code to control how sheep graze and any related actions and senses.
- Exploring: Provides any code that is used to control how agents move around the world.
- Breeding: Provides any code that controls the breeding of agents.
- Hunting: Provides the code that controls how wolves hunt and each their prey.

Each of these behaviours could contain any number of primitives, but as the needs of this model are simple, each one only contains one or two, the following is a list of the actions 'a_abc' and senses 'update_abc':

- Grazing:
  - update_standingOnGrass
  - a_eatGrass
  - a_lookForGrass
- Exploring:
  - a_wander
- Breeding:
  - a_breed
- Hunting:
  - update_nearSheep
  - a_eatSheep
  - a_lookSheep

There are several things to note when considering the above behaviour library, firstly as already mentioned 'update_' functions are optional functions that allow for more powerful senses. In the case of this model, both sheep and wolves also own the energy attribute and the planner is able to use a sense called 'energy' to incorporate that information into the

plan. As energy is updated whenever it is used for example by a_wander there is no need for an update function.

Also, this trivially simple model is a good demonstration of the modularity of good BOD code, any of the individual primitives can easily be enhanced, as well as new ones created without affecting existing plans or breaking compatibility.

Good naming of the primitives is also an important issue, although users can refer to documentation where necessary. It is conceivable that a user with little relevant experience could create a plan using these primitives without direct instructions.

## 7.3 POSH Plan

Using this Behaviour Library I then constructed two separate lap files with one containing the POSH plan for the sheep breed and the other for the wolf breed. Due to their simplistic nature I was able to complete them easily in a basic text editor, however it would also have been possible to use the ABODE IDE.

In an effort to fully test all parts of the scheduler the plans utilise most if not all of the possibilities available to POSH planners, these possibilities include:

- The use of actions patterns and competences.
- The use of senses to trigger certain actions as well as 'true' and 'false' values where needed.
- Frequency regulated plan elements. For example sheep are only able to follow the breed action pattern every 10 'ticks'.

## 7.4 Results

With the model and agents fully specified it is possible to run various different configurations to look at how BODNetLogo performs.

### 7.4.1 Reliability

The first thing to look at is the reliability of the model, can a simulation specified in BODNetLogo reliably reproduce the same results as a similar model specified only in NetLogo? This is not to prove that BODNetLogo can duplicate exactly NetLogo models, instead it helps to show that BODNetLogo does not introduce any inaccuracies or bugs to simulations.

In order to test this a variation of the model is created to match the default settings of the NetLogo 'wolf sheep predation' model. With the same size environment, grass growth characteristics, etc. For this variation I only used sheep agents and the aim is to see how

the population grows and what population the world can support. Figure 7.2 shows the results from a number of runs. As there is some random noise introduced as part of the breeding behaviours no single run is identical however it is possible to see clear relationships between a model in NetLogo and a model in BODNetLogo.

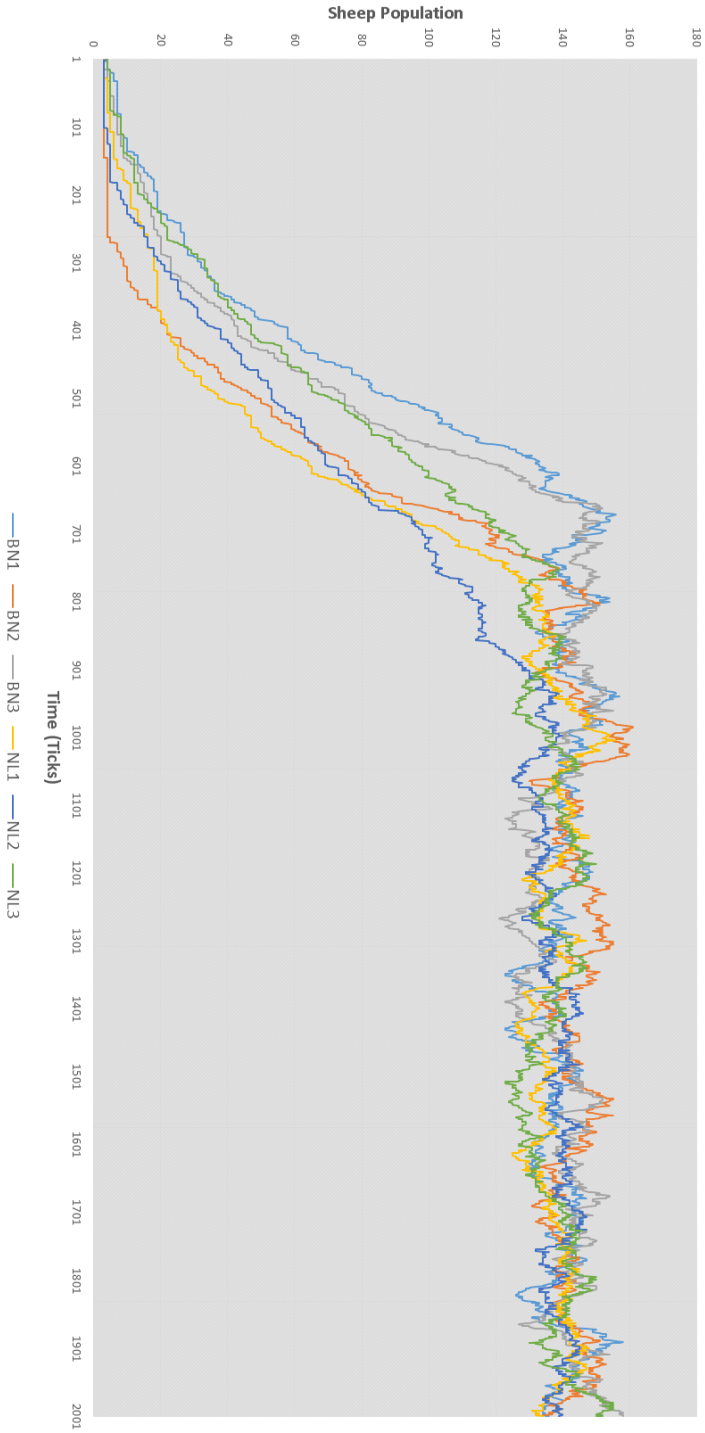


Figure 7.2: The growth and stabilisation of sheep populations. 'BN' refers to a run in BODNetLogo. 'NL' a run in NetLogo. This graph shows the ability of BODNetLogo to reproduce effects seen in NetLogo only models. The raw data for the graph can be found in Appendix B.

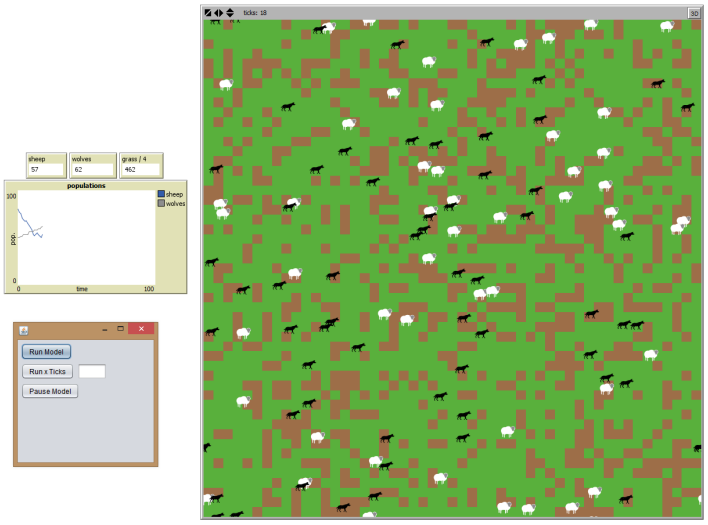


Figure 7.3: A BODNetLogo model running with both sheep and wolf agents. This shows BODNetLogo's ability to have multiple agents of different breeds operating in the same model, interacting with each other.

### 7.4.2 Flexibility

To test the flexibility of BODNetLogo, in other words the breadth of its functionality, I introduced the wolf breed to the model, this adds a number of complications. Not only are there now two different species in the NetLogo model, those species are using different action plans, have different attributes and different capabilities. As well as sheep agents being able to breed and die of natural causes (running out of energy) there are now wolves breeding and eating the sheep. Although this is a fairly specific example it is useful to demonstrate that practically any real world simulation with varied agents can be created within BODNetLogo.

Using manually calculated predictions, as well as knowledge gathered from the NetLogo version of this model it is possible to see that as with the previous configuration this model runs without error. The agents successfully interact without any unexpected behaviours or characteristics.

## 7.5 Miscellaneous Tests

In addition to the full example I have described I performed a number of much smaller tests to ensure the correctness of the various elements. The majority of these tests were in order to validate the POSH scheduler as although it was based on another implementation there were numerous changes involved in the translation. These tests included:

- Constructing plans with different combinations and orders of elements, e.g. a competence which also contains another competence and an action pattern.
- Testing the correct ranking of the hierarchy, this required constructing a basic plan and manually calculating the correct outputs, before comparing this to the output from the scheduler.
- Testing the reliability of the results by running the scheduler on a plan with differing sense inputs and checking for consistent, reliable outputs.

In addition to the scheduler, other elements of BODNetLogo required some form of correctness testing to ensure their correctness and robustness. This included the GUI which contains a number of possible bugs with data being remembered and then repopulated as users browse through the various 'levels'. An example of a bug that was found in the GUI involves the specification of the attributes. A combination of adding new attributes, removing and re-ordering elements showed that interface was sometimes mishandling the ordering of these attributes.

## 7.6 Summary

In this chapter I discussed the extended example model that I constructed as part of the testing for the system. Containing a number of very simple behaviours along with two different action plans to control different species of agents this example model demonstrates and tests a large proportion of the functionality of BODNetLogo.

I used this model to show that BODNetLogo can produce reliable results when run repeatedly with the same conditions. Also I showed the flexibility and functionality by creating a configuration with two species interacting in complex ways with the scheduler reliably keeping track and performing the actions for all the active agents.

In addition to the example model, I performed a number of other, smaller tests on individual elements of BODNetLogo. Although it is difficult to predict every possible combination of input that users will subject the program to, I believe that I have tested a representative sample of these and identified any serious flaws.

# Chapter 8

# Discussion

In this chapter I will discuss the outcome of this project. Primarily I will discuss how the results from testing (chapter 7) compare to the original objectives (chapter 3).

At a high level the project's goals have been met, I have created and demonstrated a program that allows the creation and use of complex agents using the BOD methodology. BODNetLogo gives its users the potential to create a NetLogo simulation along with a library of behaviours with actions and senses. A simulation can then be run with the agents using a POSH plan as the action selection mechanism.

## 8.1 Key Features

### 8.1.1 Complexity

The first key feature set out for BODNetLogo is that it does not add any significant complexity to the process of creating a model. It is certainly correct to note that BODNetLogo has added a number of steps to the creation and use of a model, however this does not necessarily imply an increase in complexity.

Steps such as selecting the NetLogo simulation file location or the location of the lap files for the agents are so trivial as to not contribute adversely nor favourably to the complexity for users even though they are not present in NetLogo alone. This triviality could be said to be a combination of two things. Firstly the nature of the task dictates that there should not be many new elements that have to be introduced, the program BODNetLogo's primary use is to enable the POSH action selection method whilst the other component of BOD the behaviour library is contained in the existing NetLogo interface. This means that at most BODNetLogo should ask the user for a series of configuration options. Secondly, having identified complexity as an issue I designed the interface and program in general to be as simple for the user as possible.

However there are a few elements that could be considered sub-optimal, the most obvious of

these is the need to declare the attributes of each breed not only without any typographical errors but also in the exact order that they are specified in NetLogo. This is further complicated by the fact that when declaring variables in NetLogo it can be done both globally, where every agent no matter the breed gains that attribute, but also in breed specific terms. Further it is not possible for BODNetLogo to identify incorrect ordering of elements, only a misalignment in the total number. This means that it is possible for a user to make a slight mistake that will lead to either a difficult to identify error when running the simulation or even that the simulation appears to run correctly but using faulty values for the senses of one or more agents.

The complication of the attributes was forced by the slight weakness in the NetLogo API, that is variables can only be retrieved by index value and not name, had this not been the case there would have been no need to implement the attribute store, nor ask the users to define the attributes.

Despite these few minor issues I believe that BODNetLogo does not add any complexity to the process that it would make it any more difficult to use than an average NetLogo simulation. In fact I would argue that the potential benefits of the modularity for behaviours and planning abstractions are a net improvement to the modelling process. Additionally I would add that BODNetLogo brings a further use case to the platform, as shown by Kwong (2003) and Gemrot et al. (2011) it is now possible for some task specific behaviour library to be created and then shared so that users who are unable to program can now create complex agents with only a basic understanding of the principles of POSH and some documentation that provides a list of primitives and their functionality.

### 8.1.2 BOD Capabilities

The second key feature discussed in the objectives is that BOD is a large and complex platform that allows for the specification and control of very complex agents. It is important that an integration with NetLogo does not sacrifice too much of this complexity. In section 3.1.1 I introduced one of the key compromises necessitated by the integration, that of the loss of different behaviour modules running simultaneously. This independence from the action selection methods allowed behaviours to take on complex tasks such as learning or planning so long as they provide an instant response when some action or sense is called (Bryson, 2003*a*).

Although this is indeed a large reduction in capabilities there is a strong argument to say that this functionality is not needed in this situation. Operation in parallel, whilst allowing complex learning calculations, is more often needed in order to control non-instant functionality in BOD robots, a lot of robot functionality is non-instantaneous, everything from moving to looking around requires some motor operation and a relatively long time to operate. Thus the parallel nature of BOD was used to keep a robot reactive whilst it was performing these functions, instead of telling the robot to move then waiting for completion, the action selection mechanism can tell the behaviour to move the robot and then continue executing the plan. In ABM this functionality is not needed, NetLogo provides instant

responses when an agent asks to move or needs to sense nearby agents there are no physical limitations.

My example model however showed that whilst the functionality of modular behaviours is lost, the design idea can at least be preserved. Additionally as the scheduler is completely external to NetLogo there is no practical limitation on its functionality should there be a need for future updates.

### 8.1.3 Reliable Results

Although it is very difficult to test and verify perfectly that the BODNetLogo platform is stable and reliable enough for scientifically valid results, I believe that it is fair to reason that both BOD and NetLogo have been proven and widely used for this purpose (Sklar, 2007; Gemrot et al., 2011), further, my own model albeit limited did not demonstrate any characteristics that would indicate that the integration between the two platforms had introduced any source of errors.

With this in consideration I believe that pending any contradictory evidence BODNetLogo can be said to be a suitably stable platform that does not have any inherent errors that would cause discrepancies in any produced data.

## 8.2 Design Requirements

In addition to the key features I also proposed several specific requirements that could be used to judge the performance of BODNetLogo, they are as follows:

1. Allow a POSH action plan to control an agent within a NetLogo simulation.
    1.1. Run a POSH action plan using an implementation of an action scheduler,
    1.2. Allow the action scheduler to control a NetLogo agent using the API.
2. Allow Multiple Agents to be controlled within a NetLogo simulation.
    2.1. Allow duplicates of a single POSH plan to control different agents,
    2.2. Allow multiple POSH plans to control different agents.

Now that I have a complete working example of BODNetLogo I believe I can address each of these points:

1. Yes, the wolf sheep example shows at least one agent being controlled by a POSH action plan.
    1.1. Yes, again the example shown in chapter 7 shows my own scheduler implementation interpreting and following a POSH plan,

1.2. Yes, the action scheduler via the Action and Sense classes uses the NetLogo API to send commands to the model.

2. Yes, the example shows several agents of different breeds to be in the same simulation and interact.

2.1. Yes, in this example all agents of the same breed are controlled by a single specified plan,

2.2. Yes, the different breeds are using a different plan to control their action selection.

In addition to those functional requirements I added a number of non-functional requirements that whilst not strictly necessary would be useful:

1. The application will be easily accessible to non-computer science users.

2. An example/template simulation is provided to demonstrate the correct use of BOD.

1. Although it is my belief that BODNetLogo is accessible to at least the audience of NetLogo if not wider I have not performed any user tests, and so cannot conclusively answer this point.

2. Yes, I developed a full example that provides a good representative demonstration of the features.

## 8.3 NetLogo

An important factor in the success or failure of this project is the quality of the NetLogo API. As already mentioned there is a flaw in the accessing of variables that caused some otherwise unnecessary complications. However I believe that other than that rather specific need the API is of a very high standard and very wide capabilities. One example of this quality and power is the way that the index value, once it has been calculated, is used to gather the actual value. One possible way would be to send a String request for it, and receive a String reply. However the API provides a much more powerful option, it is possible to request an agent or patch by its ID, the return value is then an *Object* of the type Agent or Patch. This object can be used to access practically any information about the agent. Whilst this implementation does not need that capability, the potential allows for future work to greatly increase the capabilities of BODNetLogo.

Another example of the quality of the NetLogo API is the *command* function, this is the equivalent to having access to the command bar at the bottom of the NetLogo GUI. With command it is possible to access any functionality (assuming no return value is needed) that is available when writing a simulation directly into NetLogo.

## 8.4 BODMASON

As already mentioned (section 2.3) a similar project to integrate BOD with MASON has already been completed (Bryson et al., 2006). MASON, like NetLogo, is an agent based modelling platform, however it has a different focus to NetLogo in that its primary design principles are speed and customisation. Rather than being aimed at the non-computer science audience as NetLogo is, MASON is targeted at experienced programmers who need faster results, a much more customised model, or both. Partly because of this but also as a result of coming from different authors with different ideas BODMASON and BODNetLogo differ in several important ways.

One of the first and most noticeable differences is that the authors of BODMASON decided to use and further the Python implementation of POSH scheduler by modifying it slightly to run as Jython code. Already the most fully featured, the authors further enhanced the scheduler with performance updates and other minor features. This is despite the fact that MASON is written largely in Java as is NetLogo. In order to make this work the authors implemented code which allows the behaviour library, also written in Jython to communicate with MASON. The side effect of this is that it is reasonably complex to create a new behaviour library for this platform, however given the audience of MASON, this was a fair trade-off.

As part of the communication with MASON from the Jython scheduler, and due to architecture of MASON, the parts of BODMASON written in Java are much more tightly integrated into the platform than BODNetLogo which only accesses NetLogo in two places via API commands.

An example of this integration is the control loop, in BODNetLogo there was no way to take control of the built in control loop, this made it necessary to recreate one of my own. In the case of BODMASON however the authors were able to create their control loop by instantiating directly their own version of the MASON loop, thereby overriding the original and replacing it with their own. Although this deep integration exposes BODMASON to incompatibilities it allows for more efficient code that does not have to work through a general API. Once again this is a trade-off that makes sense for the performance orientated users of MASON, but less so for NetLogo where simplicity is much more important.

As their project was also the first to integrate the real time BOD with a turn based modelling platform their work also has much more focus on the implications of *Synchronous Asynchrony* as they refer to it. However the work they did into investigating the implications of this possible conflict allowed me to focus less on that particular issue and more on the actual integration.

## 8.5 Future Work

Although BODNetLogo meets its primary aims it is by no means a finished product, there are several features that can be improved or added that will increase its usefulness to the users. Although these improvements are beyond the scope of this project it is useful to note some of these improvements that could be made in the future. I can break these improvements into two key areas. Improvements or fixes to existing functionality and new functionality that could be added.

### 8.5.1 Fixes and Improvements

The primary source of criticism that I found through my use of BODNetLogo in chapter 7 was the poor efficiency of the scheduler. A modest increase in the number of agents can significantly reduce the overall speed of the model. This is not simply a user interface issue however, the purpose of ABM is to provide a great deal of statistical values that can prove or disprove a theory. A poor performing scheduler limits the use case of the program as a whole. The approach to fixing this issue should be twofold, firstly and primarily the scheduler should be further improved and optimised, removing any unnecessary steps or interactions.

Secondly threading the scheduler could garner some significant improvements, although only one agent at a time can be acting it is possible that a multi core machine could be running the scheduler for not only the current agent but also the next few in the control loop. However there is an important issue that should be considered if threading is to be implemented, reactive action selection uses the current state of the world to decide the next action of an agent, if an agent is planning its next action before the previous agent in the control loop has acted then when its turn to act arrives it could be operating on information that is no longer true. This is an issue that closely relates to the issue of asynchronous synchrony as discussed by Bryson et al. (2005).

There is however a mitigating factor to consider when looking into the performance of the scheduler, it is easy to compare BODNetLogo against NetLogo in terms of performance. However the use cases for the two platforms are likely to be very different. In particular NetLogo is designed to produce intelligent group activity through the use of many very simple agents. Whereas BODNetLogo is capable of producing much more complex agents that are capable of extremely intelligent activity. In terms of examples it is possible to compare the difference in the two to the difference between simulating a bee hive and simulating a small colony of primates. This is not to say however that performance should not be increased where ever possible, instead it is just an important factor to consider when analysing the performance of the platform.

A further improvement that could be made is to limit the need for user input or even remove it entirely with regards to the AttributeStore element. If this could be achieved either by finding some work around based on current NetLogo API functionality or even some co-

operation with the NetLogo authors it could significantly reduce the amount of user input needed to specify BODNetLogo agents. Similarly any other parts of the specification that could be automated would be a gain for the user.

The configuration file functionality that was introduced in a basic way in this implementation could be improved with the options ranging from a separate file creator, to the ability to save a specified model to a configuration file.

Finally the last feature of BODNetLogo that is not currently performing in an optimal manner is the restarting of a model, currently if a model expires through the death of all the agents, or if the user wants to modify some variable the entire BODNetLogo program must be restarted, improving the control loop so that it is able to handle restarting a model would be a useful feature.

### 8.5.2 New Features

The primary missing feature from BODNetLogo is I believe the Behaviour Space functionality currently offered by NetLogo. Behaviour Space allows users to automate the running of a large number of tests on a model, this is useful for producing a large number of results. Unfortunately the replacement of the NetLogo control loop breaks this functionality. Although it may be possible to recreate this functionality it would be my preference that as with the rest of this project it should be done by reusing the current functionality. Therefore it would be best, if it is possible, to fix this functionality using some API access so that it works as the current implementation does from a user's perspective.

One of the features that I removed from my implementation of the scheduler, as it was not something that could be easily translated, is the idea of latching. Latching increases the complexity of the action selection mechanism even further, bringing it closer to what could be considered a true intelligent agent (Rohlfshagen and Bryson, 2008) by removing the possibility for 'dithering' between goals. Re-implementing this functionality would be a useful addition to the platform.

A networked version of BODNetLogo would also be useful. Specifically, convert the first part of the program, up to and including the control loop into a server configuration. Allowing 'clients' which would consist of any necessary networked code along with the current implementation of the 'Agent' class along with the drive collection. The control loop could then fire the agents and handle the responses over a network connection. The possible benefits of this are twofold. Firstly it would allow a BODNetLogo simulation to be a group or competitive environment with agents from different users working in the same world. Although this would be possible currently it would involve creating local copies of each user's agent.

The second possible benefit is performance, if it is not possible to greatly increase the speed of the scheduler, especially for very complex action plans. Then, although a network connection is relatively slow there may be possible speed ups available by running agents on different computers and communicating with a server which controls the overall model. The

interactions of such a networked model are complicated and require further research into areas including architecture and the possibilities of allowing out of turn actions. Specifically each agent is only allowed to act once per each tick but is it possible for the agents to act as soon as they are ready or is it necessary for issues such as sensing for the order to be predefined.

## 8.6 Summary

In this chapter I evaluated the success of BODNetLogo through a comparison with the original design objectives. From this evaluation I can say that BODNetLogo meets all of the original goals and thus fulfils the aim of this project: to provide a solution that allows the specification of complete complex agents using the BOD methodology to run in a NetLogo simulation.

Breaking this primary goal down further I can say that BODNetLogo meets this goal without adding any significant complexity, without reducing to a large extent the potential functionality of behaviour oriented design and can be used to produce meaningful results to test or model a hypothesis.

I also looked at the key differences between my own project and its closest counterpart, BODMASON. Although there was naturally an amount of common ground between the two projects there was a significant difference in our approaches. This can be considered to be largely due to the difference in target audiences, with BODNetLogo aimed at educational and non-experienced programmers and BODMASON aimed at experienced programmers interested in speed.

Finally, whilst it is possible to consider the wide range of possible future work as seemingly indicating the incompleteness of BODNetLogo, I believe that it is more accurate to consider it to show the potential scope for the functionality that it could come encompass.

# Chapter 9

# Conclusion

The aim of this project was to provide an integration between NetLogo and behaviour oriented design, this has the combined benefits of firstly, allowing more complex agents to be specified within NetLogo, and secondly providing a broader audience for BOD by increasing its usefulness. Specifically this integration required the creation of a new POSH scheduler to control the BOD agents through the use of a POSH action plan.

The justification for this integration was that although BOD has strong roots in robotics and reactive intelligence, it also has a growing presence in virtual reality and game AI. NetLogo is a very popular example of an agent based modelling platform that has a large audience in education and non-computer science academic research. However NetLogo does not currently implement any formal action selection method which means that action selection in models is limited to the users own understanding of the principle. BOD has a proven record in agent design and action selection which makes it a good fit to improve the functionality of NetLogo.

Considering NetLogo's current audience, ease of use was a big factor in BODNetLogo, not maintaining this usability would have significantly reduced the justification for the project. This may have led to some compromises in the large breadth of functionality offered by BOD however with careful design it was possible to limit this loss to some of the more complex and niche functionality such as concurrent behaviour modules.

As BODNetLogo is an integration work between two living platforms that are constantly growing and changing it was important to adopt a modular design that would limit the scope of potential problems caused by an updated version of one or both of BOD and NetLogo.

The implementation of BODNetLogo further emphasised the usefulness of the modular design, it was only necessary to access the NetLogo API in two places, making any chance of a break in compatibility slim, and relatively easy to correct if necessary. Additionally BODNetLogo only relies on the scheduler to return some drive collection that is capable of being fired, it is therefore possible to make changes to the scheduler for efficiency or even

for additional features by altering the firing process, as long as the output is some action there is no effect on BODNetLogo.

However the implementation was by no means a simple project, there were two main obstacles. Firstly a mismatch in design and functionality between BOD and NetLogo meant that I had to engineer some solutions that could bridge gaps in functionality, one example being the limited output of agent data using the NetLogo API and the need for senses in POSH planning.

The second major obstacle in creating BODNetLogo was the implementation of the POSH scheduler. This was a very time intensive process and although I had another implementation to use as a basis, translation is rarely a simple task. Through a combination of sheer size and complexity, large mismatches in programming language fundamentals and subtle and hard to spot differences in apparently similar ideas, the creation of my own POSH scheduler took a significant amount of time and work. Although it would have been simpler to use an existing scheduler, I maintain that the combination of a single programming language and a considered architecture gives BODNetLogo the best possible capabilities and potential.

The user interface is a relatively small part of BODNetLogo but as the only part of the program which most users will ever see it is a significant one. As intended the UI is fairly simple, leaving a large proportion of the task to the excellent interface in NetLogo.

Finally the wolf sheep model provided as both an example and as a test completed the system. It showed that BODNetLogo achieves its main and minor goals, offering a relatively straightforward way to build complex agents in an excellent simulation environment.

The work of this project and specifically that of the scheduler has also radically changed my opinion of BOD itself. When first considering this problem I believed that POSH was simply an if/else generator that translated an action plan into a basic tree. However further research and investigation of the scheduler highlighted how wrong that opinion was. POSH is a *very* powerful action selection method that no matter the size and complexity of plan can provide reactive, fast decisions. Additionally the design concepts behind the behaviour library and the heuristics for designing the agents provide a significant contribution to code that otherwise quickly becomes unclear and complex.

The work of Swen Gaudl on the C# scheduler and the authors of BODMASON made a significant contribution to my work. The C# scheduler providing a very strong basis on which I could create my own in Java, and BODMASON for addressing many of the issues that effected my own work.

As it stands BODNetLogo is a functional solution that meets the needs of its users however there are still some issues. In particular the performance of the scheduler is something which needs to be resolved

# Bibliography


Almeida, J., Kokkinogenis, Z. and Rossetti, R. (2012), Netlogo implementation of an evacuation scenario, *in* 'Information Systems and Technologies (CISTI), 2012 7th Iberian Conference on', pp. 1–4.

Blumberg, B. M. (1996), Old Tricks, New Dogs: Ethology and Interactive Creatures, PhD thesis, MIT.

Brooks, R. A. (1991*a*), 'Intelligence without representation', *Artificial Intelligence* **47**(13), 139 – 159.

Brooks, R. A. (1991*b*), 'New approaches to robotics', *Science* **253**(5025), pp. 1227–1232.

Bryson, J. (2003*a*), The behavior-oriented design of modular agent intelligence, *in* J. Carbonell, J. Siekmann, R. Kowalczyk, J. Mller, H. Tianfield and R. Unland, eds, 'Agent Technologies, Infrastructures, Tools, and Applications for E-Services', Vol. 2592 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 61–76.

Bryson, J., Caulfield, T. and Drugowitsch, J. (2005), Integrating life-like action selection into cycle-based agent simulation environments, *in* 'Proceedings of Agent', Citeseer, pp. 67–82.

Bryson, J. J. (2001), Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents, PhD thesis, MIT, Department of EECS, Cambridge, MA. AI Technical Report 2001-003.

Bryson, J. J. (2003*b*), Action selection and individuation in agent based modelling, *in* D. L. Sallach and C. Macal, eds, 'Proceedings of Agent 2003: Challenges in Social Simulation', Argonne National Laboratory, Argonne, IL, pp. 317–330.

Bryson, J. J., Caulfield, T. J. and Drugowitsch, J. (2006), Integrating life-like action selection into cycle-based agent simulation environments, *in* M. North, D. L. Sallach and C. Macal, eds, 'Proceedings of Agent 2005: Generative Social Processes, Models, and Mechanisms', Argonne National Laboratory, pp. 67–81.

Bryson, J. J. and Stein, L. A. (2001), Modularity and design in reactive intelligence, *in* 'Proceedings of the 17th International Joint Conference on Artificial Intelligence', pp. 1115–1120.

Bryson, J. and McGonigle, B. (1998), Agent architecture as object oriented design, *in* M. Singh, A. Rao and M. Wooldridge, eds, 'Intelligent Agents IV Agent Theories, Architectures, and Languages', Vol. 1365 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 15–29.

Gat, E. (1997), 'On Three-Layer Architectures'.

Gaudl, S., Davies, S. and Bryson, J. J. (2013), Behaviour oriented design for real-time-strategy games: An approach on iterative development for STARCRAFT AI, *in* 'Foundations of Digital Games (FDG)', Chania, Crete.

Gemrot, J., Brom, C., Bryson, J. J. and Bída, M. (2011), How to compare usability of techniques for the specification of virtual agents behavior? An experimental pilot study with human subjects, *in* M. Beer, C. Brom, V.-W. Soo and F. Dignum, eds, 'Proceedings of the AAMAS 2011 Workshop on the uses of Agents for Education, Games and Simulations', Taipei.

Grey, J. and Bryson, J. J. (2011), Procedural quests: A focus for agent interaction in role playing games, *in* D. Romano and D. Moffat, eds, 'Proceedings of the AISB 2011 Symposium: AI & Games', SSAISB, York, pp. 3–10.

Kwong, A. (2003), 'A Framework for Reactive Intelligence through Agile Component-Based Behaviors'.

Lozano-Perez, T., Jones, J., Mazer, E. and O'Donnell, P. (1989), 'Task-level planning of pick-and-place robot motions', *Computer* **22**(3), 21–29.

Luke, S. (2011), 'Multiagent Simulation and the MASON Library', `http://cs.gmu.edu/~eclab/projects/mason/`. MASON User Manual, Online Version 1.0.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K. and Balan, G. (2005), 'Mason: A multiagent simulation environment', *SIMULATION* **81**(7), 517–527.

Martin, B. T., Zimmer, E. I., Grimm, V. and Jager, T. (2012), 'Dynamic energy budget theory meets individual-based modelling: a generic and accessible implementation', *Methods in Ecology and Evolution* **3**(2), 445–449.

Minar, N., Burkhart, R., Langton, C. and Askenazi, M. (1996), 'The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations', *Environmental Modelling and Software* .

Moravec, H. (1983), 'The stanford cart and the cmu rover', *Proceedings of the IEEE* **71**(7), 872–884.

*NetLogo User Manual* (v5.0.3), `http://ccl.northwestern.edu/netlogo/docs/`. October 25th 2012.

North, M. J., Collier, N. T. and Vos, J. R. (2006), 'Experiences creating three implementations of the repast agent modeling toolkit', *ACM Trans. Model. Comput. Simul.* **16**(1), 1–25.

Railsback, S. F., Lytinen, S. L. and Jackson, S. K. (2006), 'Agent-based simulation platforms: Review and development recommendations', *SIMULATION* **82**(9), 609–623.

Rohlfshagen, P. and Bryson, J. J. (2008), Improved animal-like maintenance of homeostatic goals via flexible latching, *in* A. V. Samsonovich, ed., 'Proceedings of the AAAI Fall Symposium on Biologically Inspired Cognitive Architectures', AAAI Press, Arlington, VA, pp. 153–160.

Sklar, E. (2007), 'Software review: Netlogo, a multi-agent simulation environment', *Artificial Life* **13**(3), 303.

Vinatier, F., Lescourret, F., Duyck, P.-F. and Tixier, P. (2012), '"from IBM to IPM: Using individual-based models to design the spatial arrangement of traps and crops in integrated pest management strategies "', *Agriculture, Ecosystems & Environment* **146**(1), 52 – 59.

Whitehouse, H., Kahn, K., Hochberg, M. E. and Bryson, J. J. (2012), 'The role for simulations in theory construction for the social sciences: case studies concerning divergent modes of religiosity', *Religion, Brain & Behavior* **2**(3), 182–201.

Wilensky, U. (1999), 'Netlogo', `http://ccl.northwestern.edu/netlogo`. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL.

Zhou, Z., Chan, W. and Chow, J. (2007), 'Agent-based simulation of electricity markets: a survey of tools', *Artificial Intelligence Review* **28**, 305–342.

# Appendix A

# Example BODNetLogo Model

## A.1 File: wolfSheep.bnconfig

```
#Lines beginning with # are treated as comments
#This is a configuration file that is read by BODNetLogo
#Once it is read it populates the settings on the
    simulation configuration
#Window, see the comments for information on the meaning
    of each element
#
#
#To use it on your own version of BODNetLogo please
    replace the three file locations
#with the correct values for the nlogo file and the
    breed lap files
#
#
#BooleanValues
#RunSetup
true
#TICKSON
true
#DEBUGMODE
false
#AUTODEATH
true
#
#
#
#NetLogo File Location
C:\BODNetLogo\Model\wolfSheep.nlogo
#
#
#
#Start of agent declaration
#'<' character starts an agent and a '>' closes it
<
#plural name
sheep
#singular name
a-sheep
#lap file location
C:\BODNetLogo\Model\Agents\sheep.lap
# '[' indicates the start of the attributes array and a
    ']' the end, note: only 1 attribute per line
[
standingOnGrass
energy
]
>
```

```
#
#
#
<
#plural name
wolves
#singular name
wolf
#lap file location
C:\BODNetLogo\Model\Agents\wolf.lap
# '[' indicates the start of the attributes array and a
    ']' the end, note: only 1 attribute per line
[
nearSheep
energy
]
>
```

## A.2 File: wolf.lap

```
(
(AP ap-breed (minutes 1.0) (
        a_breed
))
(C c-hunt (minutes 10.0) (goal ((false)))
        (elements
                ((on-grass (trigger ((nearSheep 1.0
                    ==))) a_eatSheep))
                ((default (trigger ((true)))
                    a_lookSheep))
        )
)

(DC life (goal ((energy 0.0 <=)))
        (drives
                (

                (hungry (trigger ((energy 20.0 <=)))
                    c-hunt))
                (

                (breed (trigger ((energy 15.0 >=)))
                    ap-breed(ticks 10)))
                (

                (default-behaviour (trigger ((true)))
                    a_wander))

        )
)
)
```

## A.3 File: sheep.lap

```
(
(AP ap-breed (minutes 1.0) (
        a_breed
))
(C c-graze (minutes 10.0) (goal ((false)))
        (elements
                ((on-grass (trigger ((standingOnGrass
                    1.0 ==))) a_eatGrass))
                ((default (trigger ((true)))
                    a_lookForGrass))
        )
)

(DC life (goal ((energy 0.0 <=)))
        (drives
                (

                (hungry (trigger ((energy 5.0 <=)))
                    c-graze))
                (

                (breed (trigger ((energy 8.0 >=)))
                    ap-breed(ticks 10)))
                (

                (default-behaviour (trigger ((true)))
                    a_wander))

        )
)
```

)

## A.4 File: wolfSheep.nlogo

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BREEDS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;DEFINE BREEDS
breed [sheep a-sheep]
breed [wolves wolf]


;;DEFINE BREED ATTRIBUTES
sheep-own [standingOnGrass energy]
wolves-own [nearSheep energy]


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; SETUP
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;GLOBAL VARIABLES
globals [grass grassGrowthTime]  ;; keep track of how
    much grass there is


;;PATCH VARIABLES
patches-own [countdown]

;; SETUP FUNCTION
to setup
  ;;Ensure the simulation has been reset
  clear-all


  ;;Global values
  set grassGrowthTime 30

  ;;'Grow' the grass
  ask patches [
      set countdown random grassGrowthTime ;; initialize
          grass grow clocks randomly
      set pcolor one-of [green brown]
      ]

  ;;Set the visual shape for breeds
  set-default-shape sheep "sheep"
  set-default-shape wolves "wolf"

  ;;Define the breeds

  create-sheep 100
  ;;Define any starting values for sheep
  [
    set color white
    set size 1.5
    set label-color blue - 2
    setxy random-xcor random-ycor
    set standingOnGrass 0
    set energy 4
  ]


  create-wolves 0
  [
    set color black
    set size 1.5
    set label-color blue - 2
    setxy random-xcor random-ycor
    set nearSheep 0
    set energy 20
  ]

  ;;Any functions for the monitors
  set grass count patches with [pcolor = green]


  reset-ticks
end



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; NATURE
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to nature
```

```
  growGrass
end

to growGrass
  ask patches [
    if pcolor = brown [
      ifelse countdown <= 0
        [ set pcolor green
          set countdown grassGrowthTime ]
        [ set countdown countdown - 1 ]
    ]
  ]
  set grass count patches with [pcolor = green]

end


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BEHAVIOUR LIBRARY START
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BEHAVIOUR: GRAZING
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;SENSES;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to update_standingOnGrass
  set standingOnGrass 0
  if pcolor = green [
    set standingOnGrass 1
  ]
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;ACTIONS;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to a_eatGrass
  if pcolor = green [
    set pcolor brown
    set energy energy + 4
  ]
end

;;Currently just 'wander' but could be updated at later
    time for better behaviour
to a_lookForGrass
  rt random 50
  lt random 50
  fd 1
  set energy energy - 1
end


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BEHAVIOUR: EXPLORING
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;SENSES;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;ACTIONS;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;Wander aimlessly, turn a random angle then forward one
    tile
to a_wander
  rt random 50
  lt random 50
  fd 1
  set energy energy - 1
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BEHAVIOUR: BREEDING
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;SENSES;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;ACTIONS;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to a_breed
  if random-float 50 < 10 [
    set energy energy / 2
    hatch 1
  ]
end


to a_breed_sheep
```

```
  if random-float 50 < 50 [
    set energy energy / 2
    hatch 1
  ]
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BEHAVIOUR: HUNTING
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to update_nearSheep
  set nearSheep 0
  if count sheep in-radius 1 > 0 [
  set nearSheep 1
  ]
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

to a_eatSheep
  let prey one-of sheep in-radius 1
  if prey != nobody                                    ;; make
      sure wolf found a sheep
    [
      move-to prey
      ask prey [ die ]                                 ;; kill
          the prey
      set energy energy + 20    ;; get energy
     ]

end

;; Currently just 'wander' but could be updated at later
    time for better behaviour
to a_lookSheep
  rt random 50
  lt random 50
  fd 2
  set energy energy - 1
end

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; BEHAVIOUR LIBRARY END
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
@#$#@#$#@
GRAPHICS-WINDOW
350
10
1217
898
25
25
16.81
1
14
1
1
1
0
1
1
1
-25
25
-25
25
1
1
1
ticks
30.0

PLOT
12
312
328
509
populations
time
pop.
0.0
100.0
0.0
100.0
true
true
"" ""
PENS
```

```
"sheep" 1.0 0 -13345367 true "" "plot count sheep"
"wolves" 1.0 0 -7500403 true "" "plot count wolves"

MONITOR
50
265
121
310
sheep
count sheep
3
1
11

MONITOR
125
265
207
310
wolves
count wolves
3
1
11

MONITOR
211
265
287
310
NIL
grass / 4
0
1
11

@#$#@#$#@
@#$#@#$#@
default
true
0
Polygon -7500403 true true 150 5 40 250 150 205 260 250

airplane
true
0
Polygon -7500403 true true 150 0 135 15 120 60 120 105
    15 165 15 195 120 180 135 240 105 270 120 285 150
    270 180 285 210 270 165 240 180 180 285 195 285 165
    180 105 180 60 165 15

arrow
true
0
Polygon -7500403 true true 150 0 0 150 105 150 105 293
    195 293 195 150 300 150

box
false
0
Polygon -7500403 true true 150 285 285 225 285 75 150 135
Polygon -7500403 true true 150 135 15 75 150 15 285 75
Polygon -7500403 true true 15 75 15 225 150 285 150 135
Line -16777216 false 150 285 150 135
Line -16777216 false 150 135 15 75
Line -16777216 false 150 135 285 75

bug
true
0
Circle -7500403 true true 96 182 108
Circle -7500403 true true 110 127 80
Circle -7500403 true true 110 75 80
Line -7500403 true 150 100 80 30
Line -7500403 true 150 100 220 30

butterfly
true
0
Polygon -7500403 true true 150 165 209 199 225 225 225
    255 195 270 165 255 150 240
Polygon -7500403 true true 150 165 89 198 75 225 75 255
    105 270 135 255 150 240
Polygon -7500403 true true 139 148 100 105 55 90 25 90
    10 105 10 135 25 180 40 195 85 194 139 163
Polygon -7500403 true true 162 150 200 105 245 90 275 90
    290 105 290 135 275 180 260 195 215 195 162 165
Polygon -16777216 true false 150 255 135 225 120 150 135
    120 150 105 165 120 180 150 165 225
Circle -16777216 true false 135 90 30
```

```
Line -16777216 false 150 105 195 60
Line -16777216 false 150 105 105 60

car
false
0
Polygon -7500403 true true 300 180 279 164 261 144 240
    135 226 132 213 106 203 84 185 63 159 50 135 50 75
    60 0 150 0 165 0 225 300 225 300 180
Circle -16777216 true false 180 180 90
Circle -16777216 true false 30 180 90
Polygon -16777216 true false 162 80 132 78 134 135 209
    135 194 105 189 96 180 89
Circle -7500403 true true 47 195 58
Circle -7500403 true true 195 195 58

circle
false
0
Circle -7500403 true true 0 0 300

circle 2
false
0
Circle -7500403 true true 0 0 300
Circle -16777216 true false 30 30 240

cow
false
0
Polygon -7500403 true true 200 193 197 249 179 249 177
    196 166 187 140 189 93 191 78 179 72 211 49 209 48
    181 37 149 25 120 25 89 45 72 103 84 179 75 198 76
    252 64 272 81 293 103 285 121 255 121 242 118 224 167
Polygon -7500403 true true 73 210 86 251 62 249 48 208
Polygon -7500403 true true 25 114 16 195 9 204 23 213 25
    200 39 123

cylinder
false
0
Circle -7500403 true true 0 0 300

dot
false
0
Circle -7500403 true true 90 90 120

face happy
false
0
Circle -7500403 true true 8 8 285
Circle -16777216 true false 60 75 60
Circle -16777216 true false 180 75 60
Polygon -16777216 true false 150 255 90 239 62 213 47
    191 67 179 90 203 109 218 150 225 192 218 210 203
    227 181 251 194 236 217 212 240

face neutral
false
0
Circle -7500403 true true 8 7 285
Circle -16777216 true false 60 75 60
Circle -16777216 true false 180 75 60
Rectangle -16777216 true false 60 195 240 225

face sad
false
0
Circle -7500403 true true 8 8 285
Circle -16777216 true false 60 75 60
Circle -16777216 true false 180 75 60
Polygon -16777216 true false 150 168 90 184 62 210 47
    232 67 244 90 220 109 205 150 198 192 205 210 220
    227 242 251 229 236 206 212 183

fish
false
0
Polygon -1 true false 44 131 21 87 15 86 0 120 15 150 0
    180 13 214 20 212 45 166
Polygon -1 true false 135 195 119 235 95 218 76 210 46
    204 60 165
Polygon -1 true false 75 45 83 77 71 103 86 114 166 78
    135 60
Polygon -7500403 true true 30 136 151 77 226 81 280 119
    292 146 292 160 287 170 270 195 195 210 151 212 30
    166
Circle -16777216 true false 215 106 30
```

```
flag
false
0
Rectangle -7500403 true true 60 15 75 300
Polygon -7500403 true true 90 150 270 90 90 30
Line -7500403 true 75 135 90 135
Line -7500403 true 75 45 90 45

flower
false
0
Polygon -10899396 true false 135 120 165 165 180 210 180
    240 150 300 165 300 195 240 195 195 165 135
Circle -7500403 true true 85 132 38
Circle -7500403 true true 130 147 38
Circle -7500403 true true 192 85 38
Circle -7500403 true true 85 40 38
Circle -7500403 true true 177 40 38
Circle -7500403 true true 177 132 38
Circle -7500403 true true 70 85 38
Circle -7500403 true true 130 25 38
Circle -7500403 true true 96 51 108
Circle -16777216 true false 113 68 74
Polygon -10899396 true false 189 233 219 188 249 173 279
    188 234 218
Polygon -10899396 true false 180 255 150 210 105 210 75
    240 135 240

house
false
0
Rectangle -7500403 true true 45 120 255 285
Rectangle -16777216 true false 120 210 180 285
Polygon -7500403 true true 15 120 150 15 285 120
Line -16777216 false 30 120 270 120

leaf
false
0
Polygon -7500403 true true 150 210 135 195 120 210 60
    210 30 195 60 180 60 165 15 135 30 120 15 105 40 104
    45 90 60 90 90 105 105 120 120 120 105 60 120 60 135
    30 150 15 165 30 180 60 195 60 180 120 195 120 210
    105 240 90 255 90 263 104 285 105 270 120 285 135
    240 165 240 180 270 195 240 210 180 210 165 195
Polygon -7500403 true true 135 195 135 240 120 255 105
    255 105 285 135 285 165 240 165 195

line
true
0
Line -7500403 true 150 0 150 300

line half
true
0
Line -7500403 true 150 0 150 150

pentagon
false
0
Polygon -7500403 true true 150 15 15 120 60 285 240 285
    285 120

person
false
0
Circle -7500403 true true 110 5 80
Polygon -7500403 true true 105 90 120 195 90 285 105 300
    135 300 150 225 165 300 195 300 210 285 180 195 195
    90
Rectangle -7500403 true true 127 79 172 94
Polygon -7500403 true true 195 90 240 150 225 180 165 105
Polygon -7500403 true true 105 90 60 150 75 180 135 105

plant
false
0
Rectangle -7500403 true true 135 90 165 300
Polygon -7500403 true true 135 255 90 210 45 195 75 255
    135 285
Polygon -7500403 true true 165 255 210 210 255 195 225
    255 165 285
Polygon -7500403 true true 135 180 90 135 45 120 75 180
    135 210
Polygon -7500403 true true 165 180 165 210 225 180 255
    120 210 135
Polygon -7500403 true true 135 105 90 60 45 45 75 105
    135 135
Polygon -7500403 true true 165 105 165 135 225 105 255
```

```
    45 210 60
Polygon -7500403 true true 135 90 120 45 150 15 180 45
    165 90

sheep
false
15
Circle -1 true true 203 65 88
Circle -1 true true 70 65 162
Circle -1 true true 150 105 120
Polygon -7500403 true false 218 120 240 165 255 165 278
    120
Circle -7500403 true false 214 72 67
Rectangle -1 true true 164 223 179 298
Polygon -1 true true 45 285 30 285 30 240 15 195 45 210
Circle -1 true true 3 83 150
Rectangle -1 true true 65 221 80 296
Polygon -1 true true 195 285 210 285 210 240 240 210 195
    210
Polygon -7500403 true false 276 85 285 105 302 99 294 83
Polygon -7500403 true false 219 85 210 105 193 99 201 83

square
false
0
Rectangle -7500403 true true 30 30 270 270

square 2
false
0
Rectangle -7500403 true true 30 30 270 270
Rectangle -16777216 true false 60 60 240 240

star
false
0
Polygon -7500403 true true 151 1 185 108 298 108 207 175
    242 282 151 216 59 282 94 175 3 108 116 108

target
false
0
Circle -7500403 true true 0 0 300
Circle -16777216 true false 30 30 240
Circle -7500403 true true 60 60 180
Circle -16777216 true false 90 90 120
Circle -7500403 true true 120 120 60

tree
false
0
Circle -7500403 true true 118 3 94
Rectangle -6459832 true false 120 195 180 300
Circle -7500403 true true 65 21 108
Circle -7500403 true true 116 41 127
Circle -7500403 true true 45 90 120
Circle -7500403 true true 104 74 152

triangle
false
0
Polygon -7500403 true true 150 30 15 255 285 255

triangle 2
false
0
Polygon -7500403 true true 150 30 15 255 285 255
Polygon -16777216 true false 151 99 225 223 75 224

truck
false
0
Rectangle -7500403 true true 4 45 195 187
Polygon -7500403 true true 296 193 296 150 259 134 244
    104 208 104 207 194
Rectangle -1 true false 195 60 195 105
Polygon -16777216 true false 238 112 252 141 219 141 218
    112
Circle -16777216 true false 234 174 42
Rectangle -7500403 true true 181 185 214 194
Circle -16777216 true false 144 174 42
Circle -16777216 true false 24 174 42
Circle -7500403 false true 24 174 42
Circle -7500403 false true 144 174 42
Circle -7500403 false true 234 174 42

turtle
true
0
Polygon -10899396 true false 215 204 240 233 246 254 228
```

```
    266 215 252 193 210
Polygon -10899396 true false 195 90 225 75 245 75 260 89
    269 108 261 124 240 105 225 105 210 105
Polygon -10899396 true false 105 90 75 75 55 75 40 89 31
    108 39 124 60 105 75 105 90 105
Polygon -10899396 true false 132 85 134 64 107 51 108 17
    150 2 192 18 192 52 169 65 172 87
Polygon -10899396 true false 85 204 60 233 54 254 72 266
    85 252 107 210
Polygon -7500403 true true 119 75 179 75 209 101 224 135
    220 225 175 261 128 261 81 224 74 135 88 99

wheel
false
0
Circle -7500403 true true 3 3 294
Circle -16777216 true false 30 30 240
Line -7500403 true 150 285 150 15
Line -7500403 true 15 150 285 150
Circle -7500403 true true 120 120 60
Line -7500403 true 216 40 79 269
Line -7500403 true 40 84 269 221
Line -7500403 true 40 216 269 79
Line -7500403 true 84 40 221 269

wolf
false
0
Polygon -16777216 true false 253 133 245 131 245 133
Polygon -7500403 true true 2 194 13 197 30 191 38 193 38
    205 20 226 20 257 27 265 38 266 40 260 31 253 31 230
    60 206 68 198 75 209 66 228 65 243 82 261 84 268 100
    267 103 261 77 239 79 231 100 207 98 196 119 201 143
    202 160 195 166 210 172 213 173 238 167 251 160 248
    154 265 169 264 178 247 186 240 198 260 200 271 217
    271 219 262 207 258 195 230 192 198 210 184 227 164
    242 144 259 145 284 151 277 141 293 140 299 134 297
    127 273 119 270 105
Polygon -7500403 true true -1 195 14 180 36 166 40 153
    53 140 82 131 134 133 159 126 188 115 227 108 236
    102 238 98 268 86 269 92 281 87 269 103 269 113

x
false
0
Polygon -7500403 true true 270 75 225 30 30 225 75 270
Polygon -7500403 true true 30 75 75 30 270 225 225 270

@#$#@#$#@
NetLogo 5.0.3
@#$#@#$#@
setup
set grass? true
repeat 75 [ go ]
@#$#@#$#@
@#$#@#$#@
@#$#@#$#@
@#$#@#$#@
default
0.0
-0.2 0 1.0 0.0
0.0 1 1.0 0.0
0.2 0 1.0 0.0
link direction
true
0
Line -7500403 true 150 150 90 180
Line -7500403 true 150 150 210 180

@#$#@#$#@
0
@#$#@#$#@
```

# Appendix B

# Figure Raw Data

## B.1 Figure 7.2

Due to the very large number of data points (12,000) this data has been truncated, the included electronic data contains the full data in the file Figure72RawData, or alternatively contact the author for a copy.

| Ticks | BN1 | BN2 | BN3 | NL1 | NL2 | NL3 |
|---|---|---|---|---|---|---|
| 0 | 3 | 3 | 3 | 4 | 4 | 4 |
| 100 | 8 | 3 | 7 | 5 | 3 | 8 |
| 200 | 19 | 4 | 16 | 11 | 8 | 15 |
| 300 | 32 | 9 | 23 | 18 | 20 | 33 |
| 400 | 58 | 22 | 43 | 22 | 38 | 47 |
| 500 | 90 | 50 | 75 | 37 | 53 | 71 |
| 600 | 133 | 76 | 121 | 65 | 70 | 98 |
| 700 | 147 | 120 | 145 | 106 | 98 | 125 |
| 800 | 152 | 150 | 148 | 132 | 115 | 130 |
| 900 | 139 | 134 | 147 | 133 | 128 | 135 |
| 1000 | 145 | 158 | 140 | 152 | 138 | 135 |
| 1100 | 140 | 144 | 133 | 143 | 134 | 138 |
| 1200 | 146 | 144 | 130 | 131 | 129 | 146 |
| 1300 | 136 | 155 | 131 | 144 | 133 | 141 |
| 1400 | 138 | 138 | 128 | 132 | 144 | 140 |
| 1500 | 143 | 140 | 143 | 135 | 136 | 128 |
| 1600 | 131 | 149 | 140 | 131 | 140 | 128 |
| 1700 | 144 | 136 | 143 | 137 | 145 | 141 |
| 1800 | 137 | 144 | 143 | 141 | 139 | 148 |
| 1900 | 152 | 146 | 147 | 146 | 145 | 135 |
| 2000 | 136 | 134 | 158 | 134 | 139 | 152 |